

第一部分 预备知识

第1章 更好的C	3
1.1 两种语言简述	3
1.2 循序渐进	4
1.3 类型系统	4
1.4 函数原型	5
1.5 类型安全连接	9
1.6 引用	10
1.7 类型安全 I/O	11
1.8 标准流	12
1.9 格式化	14
1.10 操纵器	18
1.11 函数重载和函数模板	22
1.12 运算符重载	22
1.13 内联函数	24
1.14 缺省参数	25
1.15 new 和 delete	26
1.16 语句声明	26
1.17 标准库特征	27

1.18 C 的兼容性	27
1.19 小结	29
第 2 章 指针	31
2.1 容易出错的编程	31
2.2 基础	31
2.3 指针运算	35
2.4 传引用语义	38
2.5 普通指针	39
2.6 const 指针	40
2.7 指针和一维数组	42
2.8 数组作为参数	47
2.9 字符串数组	49
2.10 指针和多维数组	50
2.11 更高深的内容	53
2.12 指向函数的指针	56
2.13 指向成员函数的指针	59
2.14 封装和不完全类型	61
2.15 小结	65
第 3 章 预处理器	67
3.1 #include 指令	67
3.2 其他的预处理指令	68
3.3 预定义宏	70
3.4 条件编译	72
3.5 预处理运算符	73
3.6 实现 assert	75
3.7 宏的魅力	76
3.8 字符集、三字符运算符和双字符运算符	79
3.9 翻译阶段	82
3.10 小结	82
第 4 章 C 标准库之一：面向合格的程序员	83
4.1 <ctype.h>	84

4.2	<stdio.h>	87
4.3	<stdlib.h>	91
4.4	<string.h>	100
第5章	C 标准库之二：面向熟练的程序员	103
5.1	<assert.h>	103
5.2	<limits.h>	104
5.3	<stddef.h>	107
5.4	<time.h>	110
5.5	字符集	112
5.6	代码页	114
5.7	字符集标准	115
5.8	ISO 10646	115
5.9	统一字符编码	116
第6章	C 标准库之三：面向优秀的程序员	117
6.1	<float.h>	117
6.2	<math.h>	119
6.3	<error.h>	123
6.4	<locale.h>	124
6.5	<setjmp.h>	126
6.6	<signal.h>	127
6.7	<stdarg.h>	128
6.8	va_list 作为参数	130
6.9	应用	131
6.10	结论	135
6.11	浮点数系统	135

第二部分 主要概念

第7章	抽象	145
7.1	数据抽象	145
7.2	运算符重载	154
7.3	具体的数据类型	158

7.4 类型抽象	164
7.5 函数抽象	166
7.6 小结	167
第 8 章 模板	169
8.1 泛型编程	171
8.2 函数模板	171
8.3 类模板	173
8.4 模板参数	177
8.5 模板特化	179
8.6 小结	183
第 9 章 位操作	185
9.1 按位运算符	185
9.2 访问单独的位	187
9.3 大型置位	193
9.4 位字符串	205
9.5 Wish List	205
9.6 bitset 模板	206
9.7 vector<bool> 模板特化	208
9.8 小结	209
第 10 章 类型转换和强制类型转换	211
10.1 整数的升级	211
10.2 降级	215
10.3 算术类型转换	216
10.4 函数原型	218
10.5 显式类型转换	220
10.6 函数风格强制类型转换	221
10.7 const 的正确性	221
10.8 用户定义的类型转换	223
10.9 加强运算符[]	229
10.10 新风格强制类型转换	231
10.11 小结	233

第 11 章 可见性	235
11.1 名字中包含什么	235
11.2 作用域	235
11.3 最小的作用域	238
11.4 类的作用域	239
11.5 嵌套类	244
11.6 局部类	246
11.7 典型的名字空间	247
11.8 名字空间的作用域	249
11.9 生存期	250
11.10 临时对象的生存期	253
11.11 连接	254
11.12 类型安全连接	258
11.13 “语言”连接	258
11.14 小结	259
第 12 章 控制结构	261
12.1 结构化编程	261
12.2 分支	270
12.3 非局部分支	273
12.4 信号	277
12.5 小结	282
12.6 参考文献	282
第 13 章 异常	283
13.1 可选择的错误处理方法	283
13.2 堆栈展开	288
13.3 异常捕捉	290
13.4 标准异常	293
13.5 资源管理	293
13.6 构造函数和异常	298
13.7 内存管理	303
13.8 异常规范	304

13.9 错误处理策略	306
13.10 小结	309
第 14 章 面向对象编程	311
14.1 继承	314
14.2 不同种类的集合	316
14.3 虚函数和多态	316
14.4 抽象基类	319
14.5 实例研究：一个对象持续的框架	322
14.6 数据库访问	324
14.7 映射对象到相关模式	326
14.8 PFX 的结构	327
14.9 一个代码的预排	330
14.10 小结	349

第三部分 使用标准库

第 15 章 算法	353
15.1 复杂度	354
15.2 通用算法	356
15.3 函数对象	360
15.4 函数种类	361
15.5 函数对象适配器	362
15.6 算法种类	363
15.7 小结	366
15.8 参考文献	366
第 16 章 容器和迭代器	367
16.1 标准容器	370
16.2 迭代器	371
16.3 迭代器种类	372
16.4 特殊用途的迭代器	373
16.5 容器适配器	376
16.6 关联容器	377

16.7 应用.....	377
16.8 非标准模板库容器.....	385
16.9 小结.....	387
第 17 章 文本处理.....	389
17.1 scanf.....	389
17.2 printf.....	394
17.3 子字符串.....	397
17.4 标准 C++ 字符串类.....	403
17.5 字符串流.....	406
17.6 宽字符串.....	408
17.7 小结.....	408
第 18 章 文件处理.....	409
18.1 过滤器.....	409
18.2 二进制文件.....	412
18.3 记录处理.....	414
18.4 临时文件.....	419
18.5 可移植性.....	420
18.6 POSIX.....	420
18.7 文件描述符.....	421
18.8 通过描述符来拷贝文件.....	421
18.9 读目录条目.....	423
18.10 重定向标准错误.....	427
18.11 封装重定向操作.....	431
18.12 小结.....	436
第 19 章 时间和日期处理.....	437
19.1 Julian 日期编码.....	442
19.2 用于实际工作的日期类.....	460
19.3 计算年的星期数.....	486
19.4 小结.....	487
19.5 参考文献.....	487

第 20 章 动态内存管理	489
20.1 参差数组	489
20.2 在标准 C 中使用堆	491
20.3 C++ 的自由存储	495
20.4 浅拷贝与深拷贝	496
20.5 处理内存分配失败	499
20.6 重载 new 和 delete	499
20.7 配置 new	501
20.8 堆的管理	503
20.9 避免内存管理	504
20.10 小结	510

附 录

附录 A C/C++ 的兼容性	513
附录 B 标准 C++ 算法	515
附录 C 函数对象和适配器	525
附录 D 有注解的参考书目	529
附录 E C++ 标准的制定	531

第一部分

预备知识

更好的 C

1.1 两种语言简述

20 世纪 80 年代初期, C++起源于 AT&T, 称为带类的 C, 当时 Bjarne Stroustrup 试图用 Simula-67 编写仿真程序。“类”在 Simula 中是表示用户定义类型的术语, 编写好的仿真程序的关键是能够定义对象反映现实世界。除了把类加到 C 中使其成为最快的过程语言外, 还有什么更好的方法可以得到快速仿真呢? 选择 C 语言不仅为类提供了有效的工具, 并且也提供可移植性。虽然在 C++出现之前已经有其他语言可以通过类支持数据抽象, 但是, C++现在是应用最广泛的, 几乎每个有 C 语言编译器的主要平台都支持 C++。

第一次看 C++就可能被它不可抵抗的魅力所吸引。如果有 C 语言基础, 需要将下列术语(然后少许)增加到自己的词汇表中:

抽象类、存取限定符、适配器、(空间)分配器、基类、类、类的作用域、构造函数、复制构造函数、缺省参数、缺省构造函数、delete 运算符、派生类、析构函数、异常、异常处理器、异常特化、显式构造函数、显式特化、导出、facet、友元、函数对象、继承、内联函数、迭代器、操纵器、成员函数、成员模板、多继承、不定性、名字空间、嵌套类、new 处理器、new 运算符、新风格类型转换、一次定义规则、运算符函数、重载、局部特化、指向成员的指针、多态、私有、保护、公有、纯虚函数、引用、运行期类型识别、静态成员、流、模板、模板特化、this 指针、显著特性、try 块、类型标识、类型安全连接、using 指令、虚基类、虚析构函数、虚函数。

C++的优点在于它是一种能够处理复杂应用的强大的、高效的、面向对象的语言。因此它的缺点是它本身一定有些复杂, 并且比 C 语言掌握起来更加困难。当然 C 语言自己本身也是问题的一部分。C++是一个混合的语言, 它将面向对象特征与流行的系统编程语言混合

在一起。如果不是一个主语言绑定很少内容的话，介绍如此丰富的一组新特征是不可能的。因此与 C 语言的兼容性是 C++ 设计的一个主要目标，就像 1989 年 Bjarne 在 ANSI C++ 委员会的主题演讲中所陈述的那样，C++ 是“工程上的妥协”，并且必须要使它“越接近 C 越好，但不能过度”。

C++ 事实上是一种多范例语言，像 C 和 Pascal 那样，它支持传统的过程编程方式；像 Ada 一样，它支持数据抽象和通用性(模板)；像其他所有面向对象语言一样，它支持继承性和多态性。所有这些可能都或多或少导致了 C++ 成为“不纯”的编程语言，但是这也使 C++ 成为产品化编程中更具实践性的选择。无疑 C++ 拥有最好的性能，它可以在混合语言环境中很好地运行(不仅和 C 语言，而且也和其他语言)，并且不需要像 Smalltalk 和 LISP 运行时所需的庞大运行期资源(后者是环境的，不只是编译和连接过程)。

下面将介绍其更多的优点。

1.2 循序渐进

在没有完全掌握 C++ 的情况下也可以有效地使用它。事实上，面向对象技术承诺如果供应商为重用、可扩展性提供设计好的类库，那么建立应用程序的工作就很容易了。现有的开发环境，及其应用程序框架和可视化组件，正在兑现这一承诺。

如果觉得必须要掌握这种语言，可以一步步地去做，并且在这一过程中可以取得丰硕的成果。已出现的 3 个“顶峰”是：

1. 更好的 C；
2. 数据抽象；
3. 面向对象编程。

由于 C++ 比 C 更安全、更富于表达，所以可以将它作为一个更好的 C 使用。这个顶峰上的特征包括类型安全连接、强制的函数原型、内嵌函数、const 限定修饰符(C 从 C++ 中借用了它)，函数重载、缺省参数、引用、动态内存管理的直接语言支持。也需要意识到 C++ 和它前身之间存在着不兼容性。在这章中将探究一些使 C++ 成为更好的 C 的非面向对象的特征。因为如果不说明基于类的起源就想阐明某些更好的 C 特征是很困难的，所以我也将解释 C++ 的类机制。

1.3 类型系统

理解 C++ 最重要的部分，也许就是它对于类型安全(type safety)的贡献。上面所提及的其他面向对象语言实质上是无类型的，或最多也只能说是弱类型的，因为它们主要是在程序运行期间执行错误检查，换句话说，C++ 要求声明每个程序实体的类型，并且在编译期内它要一丝不苟地检查相同用法。正是类型安全而不是其他别的特点，使 C++ 成为更好的 C，成为常用编程工作的最合理的选择。类型系统的特征包括函数原型、类型安全连接、新风格的类

型转换、运行期类型识别 (RTTI) (有关类型转换和 RTTI 的内容请参见第 10 章)。

1.4 函数原型

在 C++ 中, 函数原型不是可选的。事实上, 在 ANSI C 委员会采用原型机制以前, 它是为 C 发明的。在你第一次使用函数前必须声明或定义每个函数, 编译器将检查每个函数调用时正确的参数数目和参数类型。此外, 在其应用时将执行自动转换。下列程序揭示一个在 C 中不使用原型时出现的普通错误。

```
/* convert1.c */
#include <stdio.h>

main(
{
    dprint(123);
    dprint(123.0);
    return 0;
}

dprint(d)
double d;          // 老式的函数定义
{
    printf("%f\n",d);
}

/* 输出:
0.000000
123.000000
*/
```

函数 `dprint` 要求带有一个 `double` 型参数, 如果不知道 `dprint` 的原型, 编译器就不知道调用 `dprint(123)` 是个错误。当为 `dprint` 提供原型时, 编译器自动将 123 变换成 `double` 型:

```
/* convert2.c */
#include <stdio.h>

void dprint(double); /*原型*/
main()
{
    dprint(123);
    dprint(123.0);
    return 0;
}

void dprint(double d)
```

```
{
    printf("%f\n",d);
}
```

```
/* 输出:
123.000000
123.000000
*/
```

除类型安全外，在 C++ 中关键的新特征是类（class），它将结构（struct）机制扩展到除了数据成员之外，还允许函数成员。与结构标记同名的一个成员函数称为构造函数，并且当声明一个对象时，它负责初始化该对象。由于 C++ 允许定义具有与系统预定义类型一样性能的数据类型，因此，对于用户自定义类型也允许隐式转换。下面的程序定义了一个新类型 A，它包含了一个 double 型的数据成员和一个带有一个 double 型参数的构造函数。

```
// convert3.cpp
#include <stdio.h>

struct A
{
    double x;
    A(double d)
    {
        printf("A::A(double)\n");
        x = d;
    }
};

void f(const A& a)
{
    printf("f: %f\n", a.x);
}

main()
{
    A a(1);
    f(a);
    f(2);
}

/* 输出:
A::A(double)
f: 1
A::A(double)
f: 2
```

由于 struct A 的构造函数期望一个 double 型参数，编译器自动地将整数 1 转换为所定义

的 `double` 型用于 `a`。在 `main` 函数的第一行调用 `f(2)` 函数产生下面的功能：

1. 将 2 转换为 `double` 型；
2. 用值 2.0 初始化一个临时的 A 对象；
3. 将对象传递给 `f`。

换句话说，编译器生成的代码等同于：

```
f(A(double(2)));
```

注意到 C++ 的函数风格的强制类型转换。表达式

```
double(2)
```

等同于

```
(double)2
```

然而，在任一转换序列里只允许有一个隐式用户定义的转换。程序清单 1.1 程序中要求用一个 B 对象去初始化一个 A 对象。B 对象转而要求一个 `double` 型，因为它惟一的构造函数是 `B::B(double)`。表达式

```
A a(1)
```

变为

```
a(B(double(1)))
```

它只有一个用户定义的转换。然而，表达式 `f(3)` 是非法的，这是因为它要求编译器提供两个自动的用户定义转换：

```
//不能隐式地既做 A 的转换又做 B 的转换
```

```
f(A(B(double(3)))) //非法
```

表达式 `f(B(3))` 是允许的，因为它显式地请求转换 `B(double(3))`，因此编译器仅提供剩余的转换到 A。

通过单一参数的构造函数的隐式转换对于混合模式表达式是很方便的。例如，标准的字符串类允许将字符串和字符数组混合，如：

```
string s1="Read my lips..."; //初始化 s1
string s2=s1+"no new taxes."; //将 s1 和常字符连接
```

程序清单 1.1 仅允许一个用户定义的转换

```
// convert4.cpp
#include <stdio.h>
```

```
struct B;
```

```
struct A
{
    double x;
    A(const B& b);
};
```

```
void f(const A& a)
{
```

```
    printf("f: %f\n", a.x);
}

struct B
{
    double y;
    B(double d) : y(d)
    {
        printf("B::B(double)\n");
    }
};

A::A(const B& b) : x(b.y)
{
    printf("A::A(const B&)\n");
}

main()
{
    A a(1);
    f(a);

    B b(2);
    f(b);

    // f(3);           // 将不编译
    f(B(3));           // 隐式 B 到 A 的变换
    f(A(4));
}

// 输出:
B::B(double)
A::A(const B&)
f: 1
B::B(double)
A::A(const B&)
f: 2
B::B(double)
A::A(const B&)
f: 3
B::B(double)
A::A(const B&)
f: 4
```

第二行等价于:

```
string s2=s1 + string("no new taxes,");
```

这是因为标准的字符串类提供了一个带有单一 `const char *` 型参数的构造函数，但有时你可能不希望编译器如此轻松；例如，假设有一个字符串构造函数带有一个单一的数字参数（其实没有），也就是说将字符串初始化为一个具体的空格数，那么下面表达式的结果将会是什么呢？

```
string s2=s1+5;
```

上式右边变为 `s1+string(5)`，意思是给 `s1` 增加 5 个空格，这多少是一个让人困惑的“特征”。你可以通过声明单参数构造函数 `explicit` 来防止这种隐式转换。由于我们假设了字符串的构造函数是这样声明的，上面的语句就是错误的形式。但是 `string s(5)` 这个声明是合法的，因为它显式地调用了构造函数，与此类似，如果用

```
explicit A (double d)
```

替换程序清单 1.3 中 A 的构造函数的声明，编译器将把表达式 `f(2)` 按错误处理。

1.5 类型安全连接

C++甚至可以通过编译单元检测出不正确的函数调用，程序清单 1.2 的程序调用了程序清单 1.3 中的一个函数。当把它作为 C 程序编译时，会得到一个错误的输出结果：

```
f: 0.000000
```

程序清单 1.2 解释程序连接（也见程序清单 1.3）

```
void f(int);
```

```
main()
{
    f(1);
}
```

程序清单 1.3 要与程序清单 1.2 连接的函数

```
#include <stdio.h>
```

```
void f(double x)
{
    printf("f: %f\n",x);
}
```

C 无法区分出函数 `f` 的不同。常规作法是把正确的函数原型放到所有编译单元都包含的头文件里。然而，在 C++ 里，一个函数的调用仅连接与之有相同标记的函数定义，即函数名称和它的参数类型顺序的组合。当作为一个 C++ 程序进行编译时，在一个流行的编译器中程序清单 1.2 和程序清单 1.3 的输出结果是：

```
Error :undefined symbol f(int) in module safel.cpp
```

大多数编译器通过把函数标记和函数一起编码来获得这种类型安全连接。这种技巧经常

称为函数名编码、名字修饰、或者（我最喜欢的）名字改编。例如，函数 `f(int)` 可能以下面的形式出现在连接器中：

```
f_Fi // f 是一个带整型参数的函数
```

但是函数 `f(double)` 则是：

```
f_Fd // f 是一个带双精度型参数的函数
```

由于名字的不同，在这个例子中连接器不能找到 `f(int)` 并报错。

1.6 引用

由于 C 函数的参数是按值传递的，若传递大型结构给函数，既费时又占用空间。大多数 C 程序员使用指针来代替按值传递，例如，如果 `struct Foo` 是一个大型记录结构，可以采用如下方法：

```
void f (struct Foo * fp)
{
    /*通过 fp 来访问 Foo 结构*/
    fp->x=...
    等等.
}
```

当然，为了使用这个函数，必须传递 `Foo` 结构的地址：

```
struct Foo a;
...
f (&a);
```

C++ 的引用机制是符号上的便捷，这样做可以减少采用指针变量的显式间接访问的烦恼。

在 C++ 上面的代码可以被描述为：

```
void f (Foo &fr)
{
    /*直接访问 Foo 的成员*/
    fr.x=...
    等等.
}
```

现在可以像这样调用 `f` 不使用地址操作符：

```
Foo a;
...
f (a);
```

`f` 原型里的 `&` 符号指导编译器通过引用来传递参数，这实际上为你处理了所有的间接访问。（对于 Pascal 程序员而言，引用参数等价于 Var 参数。）

引用调用意味着对函数参数所做的任何修改也会影响到主调程序中的原始参数。这就是说你可以编写一个实际运行的交换函数（而不是一个宏）（参见程序清单 1.4）。如果不打算修改一个引用参数，就可以像我在程序清单 1.1 中所做的那样将它声明为常引用。常引用参数

具有安全性、按值调用的符号方便性以及引用调用的有效性。

如程序清单 1.5 所示，也可以通过引用从函数中返回一个对象，在赋值语句的左边是一个函数调用，这看起来有些奇怪，但是这在运算符重载时是方便的（尤其是 `=` 和 `[]`）。

1.7 类型安全 I/O

当然每个 C 程序员都曾经使用过 `printf` 的错误格式描述符号。对 `printf` 来说没有办法检查所传递的数据项是否与字符串格式匹配。

程序清单 1.4 一个说明引用调用的交换函数

```
// swap.cpp
#include <stdio.h>

void swap(int &, int &);

main()
{
    int i = 1, j = 2;

    swap(i,j);
    printf("i == %d, j == %d\n", i, j);
}

void swap(int &x, int &y)
{
    int temp = x;
    x = y;
    y = temp;
}

//输出:
i == 2, j == 1
```

做如下事情的频率如何？仅仅是在运行时发现问题？

```
double d;
...
printf("%d\n",d);/*嘿！本应该用%f*/
换句话说，C++流库使用一个对象的类型来决定正确的格式：
double d;
...
cout<<d<<endl;//不会失败的
```

表达式 `cout<<d` 翻译成带有流和 `double` 参数的函数调用（即 `operator<<stream&,double)`，因此输出处理是不会将值传递错的。如果想用定点精度打印浮点型数字，只需这样指明一次：

```
double x = 1.5, y = 2.5;           //从现在起保留小数点后两位
cout.precision(2);                 //保持小数点后的 0
cout.setf(ios::showpoint);
cout<<x<<'\n';                    //打印 1.50
cout<<y<<'\n';                    //打印 2.50
```

1.8 标准流

C++中有 4 个预定义的流：cin（标准输入），cout（标准输出），cerr（标准错误），clog（标准错误）。除了 cerr 外其余都是全缓冲流。就像 stderr 一样，cerr 的行为好象是非缓冲的，但事实上它是单元缓冲的，也就是说它在处理完每一个对象而不是每一个字节后会自动清除缓冲。例如，带有单元缓冲的语句：

```
cerr<<"hello";
```

缓冲处理 5 个字符，然后清除缓冲区。一个非缓冲处理的流会立即发送每个字符到它的最终目的地。

程序清单 1.5 通过引用从函数中返回一个对象

```
// retref.cpp: 返回一个引用
#include <stdio.h>

int & current(); // 返回一个引用
int a[4] = {0,1,2,3};
int index = 0;

main()
{
    current() = 10;
    index = 3;
    current() = 20;
    for (int i = 0; i < 4; ++i)
        printf("%d ",a[i]);
    putchar('\n');
}

int & current()
{
    return a[index];
}

//输出:
10 1 2 20
```

下列程序将标准输入拷贝到标准输出：


```
// copy1.cpp: 将标准输入拷贝到标准输出
#include <iostream>
using namespace std;

main()
{
    char c;

    while (cin.get(c))
        cout.put(c);
}
```

注意到标准头文件名(即 `iostream`)不再使用一个 `.h` 的后缀。几乎所有 C++ 标准库中的内容, 包括流, 都驻留于名字空间 (`namespace std`) 中。一个名字空间就是一个包括声明在内的已命名的范围。上面第二行 `using` 指令指示编译器在翻译期间查找声明的名字时搜寻 `std`。标准 C 头文件也存在于 C++ 程序的 `std` 标准名字空间中, 并以字母 `c` 作为前缀。为了包含 `<stdio.h>`, 可以这样做:

```
#include < cstdio >
using namespace std;
或用通常的#include<stdio.h>。
```

一个从流中读取的函数称为提取器 (`extractor`), 而一个输出函数称为插入器 (`inserter`)。 `get` 提取器从流中把下一个字节存放到它的 `char` 引用参数中, 像多数流成员函数一样, `get` 返回流本身。当一个流出现在像上面的 `while` 循环的布尔型上下文中, 如果数据成功传递, 它检验为 `true`; 如果有错误, 则为 `false`。就像试图过了文件尾还要读文件一样。尽管这样简单的布尔型检验在大多数时间能满足, 但你可以在任何时候使用下面这些布尔型成员函数对流的状态进行询问:

```
bad ( )    严重错误 (流被误用)
fail ( )   转换错误 (数据不正确但流正常)
eof ( )    文件尾
good ( )   上述都不是
```

下例程序实现逐行拷贝:

```
// copy2.cpp: 逐行拷贝
#include <iostream>
using namespace std;

main()
{
    const size_t BUFSIZ = 128;
    char s[BUFSIZ];
    while (cin.getline(s, BUFSIZ))
        cout << s << '\n';
}
```

`getline` 提取器读取 `BUFSIZ-1` 个字符给 `s`，如果遇到一个换行符就停下来，添加一个空字节，丢弃换行符。输出流使用左移运算符作为插入器。任何对象，无论是系统预定义的还是用户自定义的，都可以是流中插入链的一部分。你必须自己重载运算符 `<<` 用于自己的类中。

程序清单 1.6 是一个说明用 `>>` 运算符来实现提取功能的程序。由于在 C 中，通常使用 `stderr` 作为提示（因为它没有被缓冲），就会在 C++ 中使用 `cerr`：

```
cerr << "Please enter an integer:" ;
cin >> i;
```

这在 C++ 中不是必需的，因为，`cout` 与 `cin` 是绑定在一起的，当输出请求输入时，一个依赖于输入流的输出流被自动地刷新。如果需要强制刷新，可以使用一个 `flush` 成员函数。

程序清单 1.6 回应值和地址的整型提示符

```
// int.cpp: 为一个整数提示
#include <iostream>
using namespace std;
```

```
main()
{
    int i;
    cout << "请输入一个整数: ";
    cin >> i;
    cout << "i == " << i << '\n';
    cout << "&i == " << &i << '\n';
}
```

//例子执行结果:

```
请输入一个整数: 10
i == 10
&i == 0xffff4
```

物理地址是以定义实现的格式打印的，通常是 16 进制，当然字符数组是个例外，打印的是字符串的值而不是地址。要想打印 C 类型字符串的地址，得把它转向 `void *`：

```
char s[ ] = ...;
cout << ( void * ) s << '\n'; // 打印地址
```

操作符 `>>` 默认方式是跳过空格。程序清单 1.7 的程序利用这个特点来计算文本文件的字数。提取字符串操作类似于 `scanf` 中的 `%s` 格式化标志。在读取字符时，也可以关闭这种跳过空格的方式（见程序清单 1.8）。

1.9 格式化

在程序清单 1.8 中 `ios::skipws` 是一个格式化标志的例子。格式化标志是位掩码值，该位掩码值可以通过成员函数 `setf` 来设置，也可用 `unsetf` 复位（参见表 1.1 的完整描述）。

程序清单 1.9 的程序阐述了数字的格式化。标准流成员函数 `precision` 用来指定浮点值显

示的小数位数。如果没有设置 `ios::showpoint` 标志, 那么末尾的零不被显示。要用前置加号来打印正数, 就用 `ios::showpos`。在上例中想要以 16 进制形式显示 `x` 和在指数形式中显示大写 `e`, 使用 `ios::uppercase`。

程序清单 1.7 计算文本文件中的字数

```
// wc.cpp: 显示字的个数
#include <iostream>
using namespace std;

main()
{
    const size_t BUFSIZ = 128;
    char s[BUFSIZ];
    size_t wc = 0;

    while (cin >> s)
        ++wc;
    cout << wc << '\n';
}

//从"wc < wc.cpp"命令输出
34
```

程序清单 1.8 与程序 `copy1.cpp` 完全相同, 但使用提取运算符读取空格

```
// copy3.cpp : 用>>读取空格符
#include <iostream>
using namespace std;

main()
{
    char c;

    //不要跳过空格符
    cin.unsetf(ios::skipws);

    while (cin >> c)
        cout << c;
}
```

表 1.1

格式化标志

标 志	含 义	默 认
<code>boolalpha</code>	以 <code>alpha</code> 格式进行布尔输入输出	
<code>showbase</code>	显示 8 进制或 16 进制前缀	<code>off</code>

续表

标 志	含 义	默 认
showpoint	显示 10 进制数末尾的零	off
showpos	正数时显示加号	off
skipws	>> 跳过空格键	on
uppercase	0X 代表 16 进制, E 代表科学计数	off
unitbuf	启用单元缓冲	off

一些格式化选项可以具有一定范围的值。例如, 用来确定显示整型数基数的 `ios::basefield` 可以被设置成 10 进制、8 进制或 16 进制。(见表 1.2 中 3 种格式化域有效的描述) 由于这些是位域而不是单个的位, 可用带两个参数形式的 `setf` 来设置。例如, 程序清单 1.10 的程序设置 8 进制数模式采用下面语句:

```
cout.setf ( ios::oct, ios::basefield );
```

用标志 `ios::showbase` 进行设置时, 8 进制以 0 开头, 16 进制以 0x 开头打印输出 (或者以 0X 开头打印输出, 如果 `ios::uppercase` 也被设置)。

程序清单 1.9 描述数据格式化

```
// float.cpp : 格式化真正的数字
#include <iostream>
using namespace std;

main()
{
    float x = 12345.6789, y = 12345;
    cout << x << ' ' << y << '\n';

    //显示两个十位数
    cout.precision(2);
    cout << x << ' ' << y << '\n';

    //显示末尾的零
    cout.setf(ios::showpoint);
    cout << x << ' ' << y << '\n';

    //显示符号
    cout.setf(ios::showpos);
    cout << x << ' ' << y << '\n';

    //返回符号和默认值的精度
    cout.unsetf(ios::showpos);
    cout.precision(0);
}
```

```
//使用科学计数法
cout.setf(ios::scientific,ios::floatfield);
float z = 1234567890.123456;
cout << z << '\n';
cout.setf(ios::uppercase);
cout << z << '\n';
}

//输出:
12345.678711 12345
12345.68 12345
12345.68 12345.00
+12345.68 +12345.00
1.234568e+09
1.234568E+09
```

表 1.2

格式化域

域	值	默认值
调整域	左、右、内部	右
基数域	十进制、八进制、十六进制	十进制
浮点域	定点、科学计数法	定点符号输出

程序清单 1.10 显示整数的基数

```
// base1.cpp : 显示整数的基数
#include <iostream>
using namespace std;

main()
{
    int x, y, z;

    cout << "输入三个整数: ";
    cin >> x >> y >> z;
    cout << x << ',' << y << ',' << z << endl;

    //在不同基数中打印
    cout << x << ',';
    cout.setf(ios::oct,ios::basefield);
    cout << y << ',';
    cout.setf(ios::hex,ios::basefield);
    cout << z << endl;
```

```
//显示基数前缀
cout.setf(ios::showbase);
cout << x << ',';
cout.setf(ios::oct,ios::basefield);
cout << y << ',';
cout.setf(ios::hex,ios::basefield);
cout << z << endl;
}
```

```
//运行结果
输入三个整数: 10 010 0x10
10,8,16
10,10,10
0xa,010,0x10
```

1.10 操纵器

当标识符 `endl` 出现在一个输出流中时, 一个换行字符就被插入并且流被刷新。标识符 `endl` 是操纵器的一个例子, 即为了副效应而插入到流的一个对象。在 `<iostream>` 中被声明的系统预定义的操纵器列于表 1.3 中。程序清单 1.11 里的程序在功能上与程序清单 1.10 的程序等价, 但它是用操纵器来代替显式调用 `setf` 函数。操纵器经常可以使代码更为高效。

表 1.3 简单的操纵器 (`<iostream>`)

操 纵 器	结果等价于 <code>setf</code> 函数	意 义
<i>[fmt flags Group]</i>		
<code>boolalpha</code>	<code>setf(boolalpha)</code>	布尔型
<code>noboolalpha</code>	<code>unsetf(boolalpha)</code>	非布尔型
<code>showbase</code>	<code>setf(showbase)</code>	显示基数
<code>noshowbase</code>	<code>unsetf(showbase)</code>	不显示基数
<code>showpoint</code>	<code>setf(showpoint)</code>	显示小数点
<code>noshowpoint</code>	<code>unsetf(showpoint)</code>	不显示小数点
<code>showpos</code>	<code>setf(showpos)</code>	正数前添加正号
<code>noshowpos</code>	<code>unsetf(showpos)</code>	正数前不添加正号
<code>skipws</code>	<code>setf(skipws)</code>	跳过输出中的空白
<code>noskipws</code>	<code>unsetf(skipws)</code>	不跳过输出中的空白

续表

操 纵 器	结果等价于 setf 函数	意义
uppercase	setf(uppercase)	大写
nouppercase	unsetf(uppercase)	小写
unitbuf	setf(unitbuf)	插入操作数后立即刷新缓冲区
nounitbuf	unsetf(unitbuf)	插入操作数后不立即刷新缓冲区
<i>[adjustfield Group]</i>		
internal	setf(internal,adjustfield)	内部对齐
left	setf(left,adjustfield)	左对齐
right	setf(right,adjustfield)	右对齐
<i>[basefield Group]</i>		
dec	setf(dec,basefield)	十进制
oct	setf(oct,basefield)	八进制
hex	setf(hex,basefield)	十六进制
<i>[float field Group]</i>		
fixed	setf(fixed,floatfield)	固定
scientific	setf(scientific,floatfield)	科学计数
<i>[other]</i>		
endl	insert a newline and calls flush()	插入一个换行符并刷新
ends	inserts a '\0'	插入一个空字符
flush	flushes the stream	强制刷新一个流

程序清单 1.11 用操纵器改变数据基数

```
// base2.cpp: 显示整数的基数
//          (使用操纵器)
#include <iostream>
using namespace std;

main()
{
    int x, y, z;
    cout << "输入三个整形数: ";
    cin >> x >> y >> z;
    cout << x << ', ' << y << ', ' << z << endl;
```

```

// 在不同基数中显示
cout << dec << x << ','
    << oct << y << ','
    << hex << z << endl;

// 显示基数前缀
cout.setf(ios::showbase);
cout << dec << x << ','
    << oct << y << ','
    << hex << z << endl;
}

```

其他的操纵器带有参数（见表 1.4）。在程序清单 1.12 中的程序是用 `setw(n)` 操纵器直接在插入序列里设置输出宽度，这样就不需要单独调用 `width`。 `ios::width` 区域是特殊的：它在每次的插入后立即重置为 0。当 `ios::width` 是 0 时，值以所需的最少字符数打印，通常，即使它们的空间不够，数字也不会被删掉。

表 1.4 参数化的操纵器（`iosmanip`）

操纵器	意义等同于
<code>resetioflags(n)</code>	复位所有 <code>n</code> 中所设置的标志（关闭 <code>n</code> 所设置的标志）
<code>setioflags(n)</code>	设置所有 <code>n</code> 所设置的标志
<code>setbase(n)</code>	等同于 <code>setf(n, ios::basefield)</code>
<code>setfill(n)</code>	等同于 <code>fill(c)</code>
<code>setprecision(n)</code>	等同于 <code>precision(n)</code>
<code>setw(n)</code>	等同于 <code>width(n)</code>

程序清单 1.12 利用 `setw` 函数设置输出域宽

```

// adjust.cpp: 调整输出
#include <iostream>
#include <iomanip>
using namespace std;

main()
{
    cout << '|' << setw(10) << "hello" << '|' << endl;

    cout.setf(ios::left, ios::adjustfield);
    cout << '|' << setw(10) << "hello" << '|' << endl;

    cout.fill('#');
}

```



```
    cout << '|' << setw(10) << "hello" << '|' << endl;
}
```

//输出:

```
| hello|
|hello |
|hello####|
```

当然你可以用操纵器

```
...<<setfill('#')<<...
```

来替换语句

```
cout.fill('#');
```

但在这种情况下,这样做似乎很不方便。

提取器通常忽视宽度设置,但C风格的字符串输入是个例外。在对字符数组进行提取操作之前应该先将域宽设置为字符数组的大小,以避免数据溢出。当处理输入行

```
nowisthetimeforall
```

时,程序清单 1.13 程序将输出:

```
nowisthet,im,eforall
```

应记住的是,编译器将空白字符默认为分隔符,所以如果输入为:

```
now is the time for all
```

那么输出将是:

```
now,is,the
```

程序清单 1.13 控制输入字符串宽度

// width.cpp: 控制输入字符串的宽度

```
#include <iostream>
```

```
#include <iomanip>
```

```
using namespace std;
```

```
main()
```

```
{
```

```
    char s1[10], s2[3], s3[20];
```

```
    cin >> setw(10) >> s1
```

```
        >> setw(3) >> s2
```

```
        >> s3;
```

```
    cout << s1 << ',' << s2 << ',' << s3 << endl;
```

```
}
```

输入输出流也支持新的布尔数据类型 (bool), 以及格式标识和用于数字或字母文本的操纵器:

```
bool b=true;
```

```
cout<<b<<endl;           //打印 "1"
```

```
cout.setf(ios::boolalpha); //或仅插入操纵器 boolalpha
```

```
cout<<b<<endl;           //打印“true”
```

你可以通过简单定义一个将流引用作为参数并返回相同引用的函数来建立一个自己的操纵器。例如，下面是一个 ASCII 码控制铃声的操纵器，当将它插入在任何输出流中时可以发出铃声：

```
//响铃操纵器
#include <iostream>
ostream& beep(ostream& os)
{
    os<<char(7); //ASCII 响铃
    return os;
}
```

使用时，只需插入：

```
cout<<...<<beep<<...
```

1.11 函数重载和函数模板

程序清单 1.4 中的交换函数 (swap) 只有在交换整数时才有用。如果要交换两个任何系统预定义的数据类型中的对象该怎么办呢？C++ 允许定义多个同名函数，只要它们的特征不同。因此就可以为所有系统预定义的数据类型定义一个交换函数：

```
void swap(char &,char &);
void swap(int &,int &);
void swap(long &,long &);
void swap(float &,float &);
void swap(double &,double &);
```

等等。

然后就可以调用交换函数用于任何两个系统预定义数据类型对象的交换。然而，假如要实现这些函数中的每一个，不用多久就会发现正在反复地做同一件事而惟一不同的是要交换对象的类型。为了使工作更简洁并减少犯低级错误，可以定义一个模板函数来代替所有的函数。关于模板详细内容请参见第 8 章。

1.12 运算符重载

在 C++ 中你可以重载运算符，例如，定义一个复数的数据类型如下：

```
struct complex
{
    double real, imag;
};
```

假如能使用中缀符号用于复数加法，那将会相当方便。如：

```
complex c1,c2;
```

```
...
complex c3=c1+c2;
```

当编译器遇到如 `c1+c2` 这样的表达式时，将查找下边两个函数中的一个（只须其中的一个存在）：

```
operator+(const complex&,const complex &);    //全局函数
complex::operator+(const complex &);          //成员函数
```

关键字 `operator` 是函数名的一部分。为实现两个复数的加法可以将 `operator+` 定义为全局类型，如下：

```
complex operator+(const complex &c1,const complex &c2)
{
    complex r;
    r.real=c1.real+c2.real;
    r.imag=c1.imag+c2.imag;
    return r;
}
```

程序清单 1.14 运算符+和运算符<<在复数中的应用

```
#include <iostream>
using namespace std;

struct complex
{
    double real, imag;
    complex(double = 0.0, double = 0.0);
};

complex::complex(double r, double i)
{
    real = r;
    imag = i;
}

inline ostream& operator<<(ostream &os, const complex &c)
{
    os << '(' << c.real << ',' << c.imag << ')';
    return os;
}

inline complex operator+(const complex &c1, const complex &c2)
{
    return complex(c1.real+c2.real,c1.imag+c2.imag);
}
```

不允许重载系统中预定义的操作，例如不允许重载两个整型数相加。因此，在重载操作中至少有一个操作数是用户自定义类型。

流库“知道”怎样通过运算符重载来格式化各种系统预定义的数据类型。例如，`ostream` 类中，`cout` 是一个实例，它为所有的系统预定义的数据类型都重载了操作符<<，当编译器看到表达式：

```
cout<<i;
```

这里 `i` 是整型，它产生以下的函数运算：

```
cout.operator<<(i); //ostream: : operator<<(ostream&,int)
```

这样可以正确地格式化数据。

程序清单 1.14 表明如何通过重载用于复数的运算符<<来扩展标准流(输出在程序清单 1.5 中)。编译器将表达式

```
cout<<c
```

转换成下面的函数调用(在这里 `c` 是一个复数)：

```
operator << ( cout , c)
```

这将依次采用 `operator<<(ostream&, const complex&)` 将操作分解成格式化系统预定义类型的对象。这个函数也返回流，因此，可以在一个单独的语句中链接多个插入流。如，表达式

```
cout<<c1<<c2
```

变为

```
operator<<(operator<<(cout,c1),c2)
```

这要求 `operator<<(ostream&, const complex&)` 返回流，为了高效这是通过引用来实现的。

1.13 内联函数

在程序清单 1.14 中所看到的关键字内联 (`inline`) 是提示编译器要把相应的代码“内联”。也就是说，直接把代码写入程序中而没有实际函数调用的开销。如果编译器准许了你的要求，它把每一次的函数调用都用相应函数的代码来代替，从而避免了实际函数调用的开销。这种机制不同于类似函数的宏，宏是在程序编译前实现文本的代换。内联函数具有实际函数的类型检查和语义，并且没有函数调用的开销和宏定义副作用的敏感性。例如，假如定义一个宏找出两个数中的较小者：

```
#define min(x,y) ((x)<(y)?(x):(y))
```

当带有增量参数时，这将是失败的，如：

```
min(x++, y++)
```

由于内联函数具有真正函数的作用，因而不会出现这种问题。

程序清单 1.15 使用复数数据类型

```
#include <iostream>
#include "complex.h"
```

```
using namespace std;

main()
{
    complex c1(1,2), c2(3,4);
    cout << c1 << " + " << c2 << " == " << c1+c2 << endl;
}

//输出:
(1,2) + (3,4) == (4,6)
```

然而并不是所有函数都可以或应该进行内联操作,当然一个递归函数是没有资格内联的,在内联时函数体大的函数会极大地增加代码量。内联主要用于代码少而简单的函数。

1.14 缺省参数

在一个函数声明中的缺省参数用于指示该函数从它的原型中取值。在程序清单 1.16 中有一个具有原型的函数:

```
int minutes(int hrs ,int min=0);
```

最后一个参数后面的“=0”指示编译器给第二个参数提供值 0。当调用 `minutes` 函数时,可以省略了该参数。这种机制对定义相应的重载函数来说本质上是一种速记的方法。在这种情况下,前面的语句等价于:

```
int minutes (int hrs,int min);
int minutes (int hrs); //忽略了minutes
```

程序清单 1.14 中的复数构造函数采用了缺省参数。允许不带参数或带有 1 个或 2 个参数来定义复数,例如:

```
complex c1; //(0,0)
complex c2(1); //(1,0)
complex c3(2, 3); //(2,3)
```

程序清单 1.14 中 `operator+` 的返回语句正是上面第三个语句。

程序清单 1.16 说明缺省参数

```
// minutes.cpp

#include <iostream>
using namespace std;

inline int minutes(int hrs, int mins = 0)
{
    return hrs * 60 + mins;
}

main()
```

```

{
    cout << "3 hrs == " << minutes(3) << " minutes" << endl;
    cout << "3 hrs, 26 min == " << minutes(3,26) << " minutes" << endl;
}

//输出:
3 hrs == 180 minutes
3 hrs, 26 min == 206 minutes

```

1.15 new 和 delete

在 C 语言中为了用堆栈，需要计算出所要创建的对象的大小：

```
struct Foo*fp = malloc(sizeof(struct Foo) );
```

在 C++ 中，运算符 `new` 用于计算出对象的大小：

```
Foo*fp=new Foo;
```

在 C 语言中分配数组，需调用不同的函数。

```
struct Foo*fpa= calloc(n,sizeof(struct Foo));
```

在 C++ 中，`new` 运算符会知道数组的大小：

```
Foo*fpa=new Foo[n];
```

此外，运算符 `new` 在返回指针之前，自动调用适当的构造函数来初始化对象。例如，

在堆栈中创建复数时，编译器会自动将它们初始化，如下：

```

complex *cp1= new complex;      // -> (0,0)
complex *cp2= new complex(1);   // -> (1,0)
complex *cp3= new complex(2,3); // -> (2,3)

```

可以用 `delete` 运算符的两种形式之一，把动态内存返还给堆栈，对于单独的对象可以这样做：

```

delete fp;
delete cp1;

```

但是，释放一个数组需要不同的句法：

```
delete [ ] fpa; // 释放数组句法
```

像 C++ 的一些其他特性一样，`new` 和 `delete` 提高了程序的类型安全性。用户不仅仅是申请一些内存，还有带有适当的类型检查和初始化的对象。如果了解更多的关于内存管理的内容，请参见第 20 章。

1.16 语句声明

在 C++ 中，声明可以出现在语句可以出现的任何地方。这就意味着不必在程序块的开始进行一组声明，而可以在第一次使用对象时定义它。例如，程序清单 1.17 中数组 `a` 在整个函数体中都是可见的，但是 `n` 直到声明后才有效，而 `i` 直到下一行才有效。注意 `i` 在第二次 `for` 循环中被再次声明，这说明了在循环中声明的变量的作用域是该循环本身。

程序清单 1.17 声明是语句

```
// declare.cpp
#include <iostream>
using namespace std;

main()
{
    int a[] = {0,1,2,3,4};

    //打印地址和大小
    cout << "a == " << (void *) a << endl;
    cout << "sizeof a == " << sizeof a << endl;

    //顺序打印
    size_t n = sizeof a / sizeof a[0];
    for (int i = 0; i < n; ++i)
        cout << a[i] << ' ';
    cout << endl;

    //倒序打印
    for (int i = n-1; i >= 0; --i)
        cout << a[i] << ' ';
    cout << endl;
}

//输出:
a == 0xffec
sizeof(a) == 10
0 1 2 3 4
4 3 2 1 0
```

1.17 标准库特征

本书的第三部分非常详细地说明了标准 C++ 库。除流外，库还提供了大量具体实用的类型和容器类。尽管在早些时候，我定义了自己的复数类型以说明类的某些特点和运算符重载，标准库还是提供了带有一系列强有力的复数数学运算的复数类型。如程序清单 1.18 所示，`complex` 是一个类模板，它可以采用任何想要的基本的数值类型(不论是浮点型、双精度型还是长双精度型)。

1.18 C 的兼容性

为了提供强类型检查和面向对象，C++ 不得不在一些语言方面与 C 不同。如果要把 C++

作为更好的 C 使用，就必须留意两种语言间的不同特性。

程序清单 1.18 说明复数模板

```
#include <iostream>
#include <complex>
using namespace std;

main()
{
    complex<double> x(1.0, 2.0), y(3.0, 4.0);

    cout << "x + y == " << x + y << endl;
    cout << "x * y == " << x * y << endl;
    cout << "conjugate of x == " << conj(x) << endl;
    cout << "normof x == " << norm(x) << endl;
}

//输出:
x + y == (4,6)
x * y == (-5,10)
conjugate of x == (1,-2)
normof x == 5
```

首先，C++ 比 C 有更多的关键字，必须避免使用表 1.5 中的任何符号作为程序的标识符。可以使用 `const` 整数对象和枚举常量定义 C++ 中数组大小，如：

```
const int SIZE=100;
enum{BIGGER=1000};
int a[SIZE], b[BIGGER];
```

全局 `const` 声明默认的是内部连接，而在 C 中它们是外部连接。这意味着在文件作用域内，可以使用 `const` 定义取代头文件中 `#define` 定义的宏，如果希望一个 `const` 对象具有外部连接特性就必须使用关键字 `extern`。

在 C 中，可以将指向任意类型的指针指向空类型 `void *` 或将指针从空类型 `void *` 指向任何其他类型，这允许你在没有指针类型转换的情况下使用 `malloc`，如：

```
#include <stdlib.h>
...
char*p=malloc (strlen(s)+1);
```

而 C++ 的类型系统不允许在转换类型的情况从一个空类型指针指向其他类型。上例中，无论如何都应当使用 `new` 运算符。

在 C 中，如果函数定义时漏掉了一些参数，编译器将不会检查你怎样使用该函数（例如，可以向它传递任何数量和类型的参数），在 C++ 中，原型 `f` 等价于 `f(void)`，如果要坚持使用不安全的 C 行为，可使用 `f(...)`。

表 1.5

C++的关键字和保留字

and	dynamic_cast	not_eq	throw
and_eq	else	operator	true
asm	enum	or	try
auto	explicit	or_eq	typedef
bitand	export	private	typename
bitor	extern	protected	typeid
bool	false	public	union
break	float	register	unsigned
case	for	reinterpret_cast	using
catch	friend	return	virtual
char	goto	short	void
class	if	signed	volatile
compl	inline	sizeof	wchar_t
const_cast	int	static	while
continue	long	static_cast	xor
default	mutable	struct	xor_eq
delete	namespace	switch	
do	new	template	
double	not	this	

最后，在 C++ 中单引用字符常量是 char 型而不是 int 型。否则，表达式

```
cout << 'a'
```

将输出内部字符码（如 ASCII 中的 97）而不是字母“a”。

要了解更多关于 C/C++ 的兼容特性，请参见附录 A。

1.19 小结

- 作为一种多范例式的语言，C++
 1. 是更好的 C；
 2. 支持数据抽象；
 3. 支持面向对象编程。
- C++ 是类型安全语言。
- 所有函数在第一次使用之前必须声明或定义。

- 引用参数直接支持引用调用语义。
- 可以重载函数和运算符。
- 模板允许创建通用函数。
- 内联函数将类似于函数的宏的高效与实际函数的安全性相结合。
- 自由存储运算符 `new` 和 `delete` 能根据类型计算对象的大小。
- 声明可以出现在函数可以出现的任意位置。

指 针

2.1 容易出错的编程

“分割违规”

“访问违规”

“可疑的指针转换”

“不可移植的指针转换”

“空指针赋值”

这些消息听起来熟悉吗？指针出错是 C++ 程序员必须应付的最令人厌恶的错误。实际上，长时间以来指针和它所提供给开发者的原始功能已经成为人们对 C 主要的批评。人们说指针太危险了。然而，C 和有点低级的 C++ 的哲学是相信程序员。对真相了解得不够以致人们认为这些语言危险，但只是一些程序员的一些借口而已。要安全有效地使用 C 和 C++，掌握指针是必要的。幸运的是，在弄清楚一些基本原则和技术之后，就会很容易地掌握指针。

2.2 基础

除了寄存器变量以外，程序中的所有对象都存储在内存中的某处，“某处”有一个地址，在开发平台上内存的每个字节的序号按顺序从 0 开始，简单地说，地址就是一个字节的顺序号。下面的程序说明了如何找到程序变量的地址：

```
// address.cpp
#include <cstdio>
#include <iostream>
using namespace std;
```

```

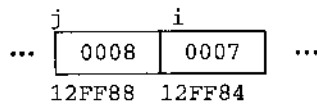
main()
{
    int i = 7, j = 8;
    printf("i == %d, &i == %p\n", i, &i);
    cout << "j == " << j << "&j == " << &j << endl;
}

/* 输出:
i == 7, &i == 0012FF88
j == 8, &j == 0x0012ff84
*/

```

单目运算符&返回一个数据对象的地址。%p 格式控制符根据编译器格式的不同来显示地址，通常是十六进制形式。本书所有的例子中，整数和地址都是 32 位（你的输出可能有所不同）的。

在上面的程序中 i 和 j 的内存分布如下：



i 和 j 在内存中碰巧相邻存储这一点并不重要（在某些结构中因对齐的需要而在对象之间存在间隔），注意我的编译器在内存中把 i 分配在 j 之后（即 i 的地址比 j 的高），这同样不重要。计算机在实际中如何存储一个数的各个位也是随系统而定的，事实上，在 PC 机中，i 中的 7 不是真正存储在 i 所占内存的最右端。我们可以假设多数情况不是这样的，因为，不管这些位在物理上如何排列，7 在逻辑上都是 0X00000007。

指针只不过是一个容纳另一个程序实体地址的变量。在大多数情况下，我们不关心一个地址的实际数值，通常只是用它引用感兴趣的对象，程序清单 2.1 中的程序说明了指针的使用。由于指针总是指向某种类型的对象，因此被引用类型通常必须出现在声明中，这样，我们说“指向整型”或者“指向字符型”等等。声明

```
int *ip;
```

说明了 *ip 是一个整型变量，因此 ip 是一个指向整型变量的指针。在声明语句之外的表达式中如果指针变量前有星号，结果表示指针变量所指向变量的值。通过指针间接地指向内存的过程叫做间接访问或复引用。它可以出现在赋值语句的任何一边。因此在程序清单 2.1 中的声明，语句

```
*ip = 9;
```

等效于

```
i = 9;
```

程序清单 2.1 说明指针和间接访问

```

// pointer.cpp
#include <iostream>
using namespace std;

```

```

main()
{
    int i = 7, j = 8;
    int* ip = &i;
    int* jp = &j;

    cout << "Address " << ip
         << " contains " << *ip << endl;
    cout << "Address " << jp
         << " contains " << *jp << endl;

    *ip = 9;
    cout << "Now Address " << ip
         << " contains " << i << endl;
    *jp = 10;
    cout << "Now Address " << jp
         << " contains " << j << endl;
}

```

//输出:

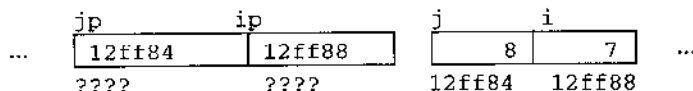
```

Address 0x0012ff88 contains 7
Address 0x0012ff84 contains 8
Now Address 0x0012ff88 contains 9
Now Address 0x0012ff84 contains 10

```

如果在定义指针时没有说明被引用的类型,表达式`*ip`将没有意义,间接访问也是不可能的。任何时候都不要忘记指针不仅仅是指向内存中的某处,它还指向某种类型的实体。这个规则的唯一例外是指针不指向任何地方的时候。当给一个指针赋特殊的值0时,就会发生这种情况(这时的指针叫做空指针,在C中由宏NULL表示)。不能复引用一个空指针,只能将它与其他指针做比较。

程序清单2.1的内存分布是:



尽管,地址通常以数字的形式出现,但是不要假设指针类型和整型数据类型之间存在任何关系。指针是一种独特的数据类型并且应该作为一种独特的数据类型来看待。只能对指针进行如下操作:

1. 在指针中存储和从其中读取被引用类型的对象的地址;
2. 改变或读取该地址中的内容(间接访问);
3. 在指针上加或减一个整数(仅限在数组中使用);
4. 与另一个指针相减或做比较(当两个指针都指向同一数组时);
5. 给指针赋值或把它和空指针做比较;

6. 作为参数传递给函数，该函数期望一个指向引用类型的指针作为参数。

由于对象的相应内存位置是不重要的（当然，数组元素除外），通常描述内存的逻辑分布更好，如：



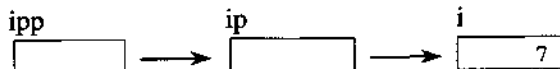
通常说法是“ip 指向 i”或“jp 指向 j”。

指针的概念如此简单以至于初学者往往由于要寻找星号之外的意义而使他们陷于紧张不安之中。如果想避免挫折和困惑而浪费没有必要的时间，只需记住：

重要的指针原则 1：指针是一个地址

注意，上面所说的是“指针是一个地址”而不是“指针保存一个地址”，当然，两种说法都是正确的。指针是地址就像整型是整数一样。人们经常不说整型“保存”一个整数。我仅仅想强调的是当使用指针时要想到它是“地址”，还有什么更简单的说法呢？

由于所有的对象都有一个地址，因此可以定义指向任何类型对象的指针，包括另一个指针对象。程序清单 2.2 中的程序说明了如何定义指向整型指针的指针。这个程序的拓扑结构是：



如果 `ipp` 是指向一个指向整型变量指针的指针，那么 `*ipp` 是该指针所指向的指针，为了最终得到 `i` 中的整数 7，需要另一级别的间接访问，即表达式 `**ipp`，换句话说：

`**ipp==*ip==i; //事实上，全部指向同一对象`

程序清单 2.2 说明指向指针的指针

```

/* ptr2ptr.cpp: 指向指针的指针 */
#include <iostream>
using namespace std;

main()
{
    int    i = 7;
    int*   ip = &i;
    int**  ipp = &ip;

    cout << "Address " << ip << " contains " << *ip << endl;
    cout << "Address " << ipp << " contains " << *ipp << endl;
    cout << "***ipp == " << **ipp << endl;
}

//输出:
Address 0x0012ff88 contains 7
Address 0x0012ff84 contains 0x0012ff88
  
```

```
**ipp == 7
```

2.3 指针运算

当一个指针指向一个数组元素时，可以在指针上加一个整数或减一个整数来使它指向同一数组的元素。在这样的指针上加 1 是通过它所引用类型的数据的字节数来增加它的值，因此，它就指向下一个数组元素。程序清单 2.3 中的程序实现了在一个浮点型数组内完成整数的运算。数组看起来像：

a		
1.0	2.0	3.0
12ff80		

在我的系统平台上，浮点型占 4 个字节。在 `p` 上加 1 实际上等于在它的值上加 4。对两个指向数组元素的指针进行相减可以得到在两个地址之间数组元素个数。换句话说，如果 `p` 和 `q` 是指向相同类型的指针，那么语句 `q=p+n` 意味着 `q-p == n`，或相反。存储两个指针差的便捷方法是把这个差存储在 `ptrdiff_t` 中，它在 `stddef.h` 中定义，包含在 `<iostream.h>` 中。

指针运算规则可以归纳为下面的公式：

1. $p \pm n == (\text{char}^*)p \pm n * \text{sizeof}(*p)$

程序清单 2.3 说明指针运算

```
/* arith.cpp: 举例说明指针运算 */
#include <iostream>
using namespace std;

main()
{
    float a[] = {1.0, 2.0, 3.0};

    //增加一个指针
    cout << "sizeof(float) == " << sizeof(float) << endl;
    float* p = &a[0];
    cout << "p == " << p << ", *p == " << *p << endl;
    ++p;
    cout << "p == " << p << ", *p == " << *p << endl;

    //减去两个指针
    ptrdiff_t diff = (p+1) - p;
    cout << "diff == " << diff << endl;
    diff = (char *) (p+1) - (char *) p;
    cout << "diff == " << diff << endl;
}

//输出:
sizeof(float) == 4
```

```

p == 0x0012ff80, *p == 1
p == 0x0012ff84, *p == 2
diff == 1
diff == 4

```

这也就是说“在（从）一个指针上加（减）一个整数 n 时，指针将在内存中向上（下）移动其所指向的类型的 n 个单元”。

2. $p-q == \pm n$

这里 n 是 p 和 q 之间元素的个数。注意，记住公式 2 中的 n 是一个特殊的类型 (`ptrdiff_t`)，在一些结构中，你只可以在指针运算中把它作为补充使用，而不能以其他的方式使用（例如，甚至不能打印它）。

因为公式假设一组有序等大小的对象，所以指针运算只在数组内有意义。然而，可以把任何单一的对象解释成字节数组。程序清单 2.4 中的程序通过把整数的地址存放在一个指向字符型的指针中来详细研究了整数，然后通过指针运算来访问每个字节。注意在 `cp` 初始化时的强制类型转换，在给不同类型的指针赋值时，需要有强制类型转换以使编译器确认你知道自己在做什么，否则编译器将怀疑你不知道自己在做什么，因此给出警告消息“可疑的指针转换”，然而，当转换成 `void` 指针时则不需要强制类型转换（参见 2.5 节“普通指针”）。

程序清单 2.4 说明指针转换

```

// convert.cpp: char* 和指针映射
#include <iostream>
using namespace std;

main()
{
    int i = 7;
    char* cp = (char*) &i;

    cout << "The integer at " << &i
         << " == " << i << endl;

    //分别打印每个字节的值:
    for (int n = 0; n < sizeof i; ++n)
        cout << "The byte at " << (void*)(cp + n)
             << " == " << int(*(cp+n)) << endl;
}

//输出:
The integer at 0x0012ff88 == 7
The byte at 0x0012ff88 == 7
The byte at 0x0012ff89 == 0
The byte at 0x0012ff8a == 0
The byte at 0x0012ff8b == 0

```


程序清单 2.4 的输出揭示了一个有趣的事实：我的 PC 的 Intel 处理器是“从后”存储的，在这种方式下一个对象最没有意义的值被存储在内存地址较低的单元中。这种存储机制叫做“不重要的值结尾”，因为在内存中向上移动时，首先遇到的是一个多字节整数的“小的结尾”。VAX 机器同样是“不重要的值结尾”。但是 IBM 机器是“重要的值结尾”。这在通常的数据处理中并不值得关注的，但是有些时候它将起很大作用。

例如，假设你想有效地存储一个世纪内的日期，需要这样存储：

```
Year(0-99)           7 bits
Month(1-12)          4 bits
Day(1-31)             5 bits
```

幸运的是，合在一起是 16 位，刚好是 PC 机中一个短整型数的大小。那么，一个将日期存储在短整型中的明显方式是用位段操作如下：

```
// bit1.cpp: 向整数中压缩数据
#include <iostream>
#include <iomanip>
using namespace std;

main()
{
    unsigned short date, year = 92, mon = 8, day = 2;

    date = (year << 9) | (mon << 5) | day;
    cout << hex << date << endl;
}

// 输出:
b902
```

日期 1992 年 8 月 2 日 (b902) 的位逻辑排列的理论期望是：

1011100	1000	00010
(92)	(8)	(2)

但是，“不重要的值结尾”机器物理上从后向前存取数据，即：

01000	0001	0011101
-------	------	---------

用下面的位域结构可得到一个可读性更强的程序（参见程序清单 2.5）。

```
struct Date
{
    unsigned day:5;
    unsigned mon:4;
    unsigned year:7;
};
```

这个结构反映了相反的排列。要用位域结构来表示一个整型，只需简单地将指向整型的

指针强制转换成指向 `Date` 的指针。现在可以不用移位和屏蔽而用名字来访问日期成员。为了通过指针访问结构成员需要先对指针进行复引用，然后再命名成员：

```
(*dp).mon
```

由于这是个繁琐的句法，其简化的句法为：

```
dp->mon
```

程序清单 2.5 通过一个位域结构封装短整型数据

```
// bit2.cpp: 用一个位域结构覆盖一个整数
#include <iostream>
#include <iomanip>

struct Date
{
    unsigned day: 5;
    unsigned mon: 4;
    unsigned year: 7;
};

main()
{
    unsigned short date, year = 92, mon = 8, day = 2;
    Date* dp = (Date*) &date;

    dp->mon = mon;
    dp->day = day;
    dp->year = year;
    cout << hex << date << endl;
}

//输出:
b902
```

我所听过的读法有“`dp` 箭头 `mon`”、“`dp` 指向 `mon`”以及一些其他的读法，在此不再提及。

2.4 传引用语义

除非被告知用别的方法，否则 C++ 总是通过值向函数传递参数。这意味着函数是局部地使用了每一个参数的拷贝。这种传递方式的结果是一个函数不可能在所调用的程序中改变对应的实参值。考虑下面的试图交换两个整型变量值的程序段：

```
void swap(int x, int y)
{
    int temp = x;
```

```

    x = y;
    y = temp;
}

```

诸如 `swap(a,b)` 这样的调用对于 `a` 和 `b` 都不会产生任何效果。在退出函数后允许改变函数固定实参值的方式叫做“传递引用”。

程序清单 2.6 用指针交换函数中实参的值

```

// swap2.cpp: 一个有用的交换函数
#include <iostream>
using namespace std;

void swap(int*, int*);

main()
{
    int i = 7, j = 8;

    swap(&i,&j);
    cout << "i == " << i << ", j == " << j << endl;
}

void swap(int* xp, int* yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

//输出:
i == 8, j == 7

```

在 C 语言中，可以通过传递想要改变其值的参数的指针来仿真传引用语义。可以通过指针，改变在调用程序中的变量值（参见程序清单 2.6）。传递引用对于大型对象和在面向对象系统中是很普通的，在面向对象操作中，指向一个对象的指针称为“句柄”。正如第 1 章中所解释的那样，通过引用，C++ 也直接支持传递引用。

2.5 普通指针

通常编写能接收指向任意类型参数的函数是很方便的。这是很有必要的，例如，用标准的库函数 `memcpy`，能够从一个地址向另一个地址拷贝一块内存。你也可能想调用 `memcpy` 来拷贝自己创建的结构：

```

struct mystruct a,b;
/*...*/
memcpy(&a,&b,sizeof(struct mystruct));

```

为了操作任意类型的指针，`memcpy` 把它头两个参数声明为 `void` 型指针。可以不需要强制类型转换将任何类型的指针赋予 `void*` 类型。也可以在 C 而不是在 C++ 中将 `void*` 赋予其他任何类型的指针。这里说明了 `void` 指针的 `memcpy` 函数的简洁实现：

```

void* memcpy(void* target, const void* source, size_t n)
{
    char* targetp = (char*) target;
    const char* sourcep = (const char*) source;
    while (n--)
        *targetp++ = *sourcep++;
    return target;
}

```

这个版本的 `memcpy` 必须把指向 `void` 的指针赋予指向 `char` 的指针，这样它就可以每次传递内存块的一个字节，并对这个字节中的数据进行拷贝。试图复引用一个 `void*` 是没有意义的，因为它的大小是未知的。

2.6 const 指针

注意 `memcpy` 函数第二个参数中的 `const` 关键字。这个关键字告诉编译器此函数将不会改变 `source` 指向的任何值（除了强制类型转换）。当把指针作为参数传递时，总是合适地使用 `const` 限定符是一个很好的习惯，它不仅可以防止你无意中错误的赋值，而且还可以防止在作为参数将指针传递给函数时可能会修改了本不想改变的指针所指向的对象的值。例如，如果在程序清单 2.6 中的声明是：

```

const int i=7,j=8;

```

有可能因为下面这条语句而得到警告：

```

swap(&i,&j);

```

因为 `swap` 的确改变了其参数所指的变量的值。

如果浏览一下你所使用的编译器提供的标准头文件，就会看到 `const`。在一个声明中，当 `const` 出现在星号前面的任意位置时，表明所指向的内容不会被改变：

```

const char *p; //指向常字符的指针
char const *q; //同样是指向常字符的指针
char c;
c=*p;          //OK(假设 p 和 q 已经初始化了)
*q=c;          //错误，不能改变指针所指的内容

```

也可以通过把 `const` 放在星号的后面来声明指针本身不可改变：

```

char* const p;
*p='a';        //OK，只有指针是常量
++p;           //错误，不能改变指针

```

要禁止改变指针和它所引用的内容，就在星号前和后都使用 `const`：

```
const char* const p;
char c;
c=*p;           //OK-能读出内容
*p='a';         //错误
++p;            //错误
```

程序清单 2.7 中的函数 `inspect` 说明了如何打印出任何对象的不同字节。因为我并没有改变对象的内容，所以第一个参数是一个指向 `const` 的指针，而且，在使用它之前要小心地将它转换成一个指向常字符的指针。

程序清单 2.7 检查任何对象的函数

```
// inspect.cpp: 检查对象的字节
#include <iostream>
#include <iomanip>
using namespace std;
void inspect(const void* ptr, size_t nbytes)
{
    const unsigned char* p = (const unsigned char*) ptr;

    cout.setf(ios::hex, ios::basefield);
    for (int i = 0; i < nbytes; ++i)
        cout << "byte " << setw(2) << setfill(' ') << i
            << ": " << setw(2) << setfill('0') << int(p[i])
            << endl;
}

main()
{
    char c = 'a';
    short i = 100;
    long n = 100000L;
    double pi = 3.141529;
    char s[] = "hello";

    inspect(&c, sizeof c);  cout << endl;
    inspect(&i, sizeof i);  cout << endl;
    inspect(&n, sizeof n);  cout << endl;
    inspect(&pi, sizeof pi); cout << endl;
    inspect(s, sizeof s);   cout << endl;
}

//输出:
byte 0:   61

byte 0:   64
byte 1:   00
```

```
byte 0:  a0
byte 1:  86
byte 2:  01
byte 3:  00
```

```
byte 0:  13
byte 1:  7c
byte 2:  d3
byte 3:  f4
byte 4:  d9
byte 5:  21
byte 6:  09
byte 7:  40
```

```
byte 0:  68
byte 1:  65
byte 2:  6c
byte 3:  6c
byte 4:  6f
byte 5:  00
```

2.7 指针和一维数组

在程序清单 2.7 中，会注意到在传递数组 `s` 时并没有使用它的地址，这是因为 C 和 C++ 在大多数表达式中把数组名转换成指向它第一个元素的指针。自 1984 年以来，我已经向成百上千的学生讲授了 C 和 C++，我注意到了指针和数组，特别是指针和多维数组之间的关系造成很多迷惑。

这样说似乎很奇怪，但是 C++ 确实不支持数组，至少 C++ 不像支持第一类数据类型如整型或者甚至结构体那样支持数组。考虑以下的语句：

```
int i=1,j;
int a[4]={0,1,,2,3},b[4];
struct pair {int j,int y;};
pair p={1,2},q;
j=i;           //OK:整型赋值
q=p;           //OK:结构体赋值
b=a;           //不能这样做
```

并不是所有有关数组的操作都是合法的。我们可以进行以下操作，但是它并不是一个“真实”的赋值：

```
int a[4]={0,1,2,3},*p;
p=a;           /*只在 p 中存储了 a[0] 的地址*/
```

除了在声明中或者当一个数组名是 `sizeof` 运算符或 `&` 运算符的操作数之外, 编译器总是把数组名解释成指向它的第一个元素的指针。可以将这个原则表达为:

```
a==&a[0]
```

或者等价于:

```
*a==a[0]
```

使用指针运算的规则, 那么当把一个整型变量 `i` 和一个数组名相加, 结果就得到指向数组第 `i` 个元素的指针, 也就是:

```
a+i==&a[i]
```

或者, 像我喜欢的表达方式一样:

重要的指针原则 2: `*(a+i)==a[i]`

程序清单 2.8 中的程序阐述了原则 2 以及准备步骤。

由于所有的数组下标是真正的指针运算, 可以使用表达式 `i[a]` 代替 `a[i]`。这些可直接从原则 2 中得到:

```
a[i]==*(a+i)==*(i+a)==i[a]
```

当然, 任何使用了这样极端错误的表达的程序都会被中断而不被执行, 而且程序员也会受到严厉的谴责。然而, 使用相反的下标也不是完全没有道理, 如果一个指针 `p` 传递一个数组, 就可以使用表达式 `p[-1]` 重新得到在 `*p` 之前的元素, 由于:

```
p[-1]==*(p-1)
```

程序清单 2.9 极为全面地涵盖了指针和数组符号的结合, 它也使用了一个关于数组中元素个数的有用公式:

```
size_t n=sizeof a/sizeof a[0];
```

虽然可以在除数中使用任何一个有效的下标, 但 0 是最安全的, 这是因为每个数组都有第 0 个元素。当然, 这一习惯只有当原始的数组声明是在生存期内才适用。

对于那些愿意使用 C 风格字符串的人来说, 一个遵循指针和数组符号概念之间相互作用的常用习惯是:

```
strncpy(s,t,n)[n]='\0';
```

程序清单 2.8 说明数组名是指针

```
// array1.cpp: 用一个数组名作为一个指针
#include <iostream>
using namespace std;

main()
{
    int a[] = {0,1,2,3,4};
    int* p = a;
```

```
    cout << "sizeof a == " << sizeof a << endl;
    cout << "sizeof p == " << sizeof p << endl;
    cout << "p == " << p << ", &a[0] == " << &a[0] << endl;
    cout << "*p == " << *p << ", a[0] == " << a[0] << endl;

    p = a + 2;
    cout << "p == " << p << ", &a[2] == " << &a[2] << endl;
    cout << "*p == " << *p << ", a[2] == " << a[2] << endl;
}

//输出:
sizeof a == 20
sizeof p == 4
p == 0x0012ff78, &a[0] == 0x0012ff78
*p == 0, a[0] == 0
p == 0x0012ff80, &a[2] == 0x0012ff80
*p == 2, a[2] == 2
```

程序清单 2.9 使用索引和指针传递数组

```
// array2.cpp: 使用索引和指针传递数组
#include <iostream>
using namespace std;

main()
{
    int a[] = {0,1,2,3,4};
    size_t n = sizeof a / sizeof a[0];

    //使用数组索引打印
    for (int i = 0; i < n; ++i)
        cout << a[i] << ' ';
    cout << endl;
    //你甚至可以交替 a 和 i (但自己别这么做!)
    for (int i = 0; i < n; ++i)
        cout << i[a] << ' ';
    cout << endl;

    //使用指针打印
    int* p = a;
    while (p < a+n)
        cout << *p++ << ' ';
    cout << endl;

    //和指针一起使用索引符是好的:
```



```

    p = a;
    for (int i = 0; i < n; ++i)
        cout << p[i] << ' ';
    cout << endl;

    //和数组一起使用指针符是好的:
    for (int i = 0; i < n; ++i)
        cout << *(a+i) << ' ';
    cout << endl;

    //使用指针向后打印:
    p = a + n-1;
    while (p >= a)
        cout << *p-- << ' ';
    cout << endl;

    //写在下方的负数是允许的:
    p = a + n-1;
    for (int i = 0; i < n; ++i)
        cout << p[-i] << ' ';
    cout << endl;
}

```

```

//输出:
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
0 1 2 3 4
4 3 2 1 0
4 3 2 1 0

```

这就把一个字符串拷贝到另一个字符串，同时确保没有溢出并且字符串没有被划上界限（假设 `n` 没有超限）——所有这些都在一个简短的语句当中实现。

在指针和数组名之间有另一个区别需要记住：一个数组名是一个不可改变的左值。这就意味着不能改变数组名对应的地址就像下面的示例所尝试的那样：

```

int a[5], b[5], *p;
/*下面所有的都不合法*/
a++;
a=p+5;
b=a;

```

如果这样赋值，就会很轻易地丢失数组在内存中存储的位置（这可不是个好主意！）。

从字面上来说字符串是一组没有名称的字符，可以使用 `sizeof` 得到它们的大小并且甚至可以给它们添加下标（见程序清单 2.10 和 2.11）。注意在清单 2.10 中我的编译器把“hello”

的每一次的出现都作为一个独立的对象，每次都返回不同的地址，有些编译器能够把具有相同字符的字符串“集中起来”以单个形式出现以节省空间。

程序清单 2.10 说明一个字符串中的字符是一个匿名数组

```
// array3.cpp
#include <iostream>
using namespace std;

main()
{
    char a[] = "hello";
    char* p = a;

    cout << "a == " << &a << ", sizeof a == " << sizeof a << endl;
    cout << "p == " << (void*)p << ", sizeof p == " << sizeof p << endl;
    cout << "sizeof \"hello\" == " << sizeof "hello" << endl;
    cout << "address of \"hello\" == " << (void*)"hello" << endl;
    cout << "address of \"hello\" == " << (void*)"hello" << endl;
}

//输出:
a == 0x0012ff84, sizeof a == 6
p == 0x0012ff84, sizeof p == 4
sizeof "hello" == 6
address of "hello" == 0x004090d4
address of "hello" == 0x004090f1
```

程序清单 2.11 将字符串中的字符进行索引

```
// array4.cpp: 将字符串中的字符索引
#include <iostream>
using namespace std;

main()
{
    for (int i = 0; i < 10; i += 2)
        cout << "0123456789"[i];
}

//输出:
02468
```

练习 2.1

已知如下声明:

```
int a[ ] = { 10, 15, 4, 25, 3, -4 };
```

```
int *p = &a[ 2 ];
下面表达式的结果是什么?
a. *(p+1)
b. p[-1]
c. p-a
d. a[*p++]
e. *(a+a[2])
```

2.8 数组作为参数

当你把数组作为参数传递给一个函数，正如所预期的那样，是传递了指向数组第一个元素的指针。因此，可以在调用的函数中永久地改变数组元素的值。在程序清单 2.12 的函数 `f` 中，地址 `&a[0]` 按值传递给指针 `b`，因此表达式 `b[i]` 就和表达式 `a[i]` 完全是一样的了。不可能按值传一个完整的内置数组。

即使用数组符号定义了参数 `b`，即：

```
int b[]
它同下面这种写法是完全一样的。
int *b
```

程序清单 2.12 说明作为参数的数组实际上是指针

```
// array5.cpp: 数组作为参数
#include <iostream>
using namespace std;

void f(int b[], size_t n)
{
    cout << "\n*** Entering function f() ***\n";
    cout << "b == " << b << endl;
    cout << "sizeof b == " << sizeof b << endl;
    for (int i = 0; i < n; ++i)
        cout << b[i] << ' ';
    b[2] = 99;
    cout << "\n*** Leaving function f() ***\n\n";
}

main()
{
    int a[] = {0,1,2,3,4};
    size_t n = sizeof a / sizeof a[0];

    cout << "a == " << a << endl;
    cout << "sizeof a == " << sizeof a << endl;
```

```

    f(a,n);
    for (int i = 0; i < n; ++i)
        cout << a[i] << ' ';
}

```

//输出:

```

a == 0x0012ff78
sizeof a == 20

```

```

*** Entering function f() ***
b == 0x0012ff78
sizeof b == 4
0 1 2 3 4
*** Leaving function f() ***

0 1 99 3 4

```

而且, `sizeof(b) == 4`, 这是我的操作平台上指针的大小。我们无法在另一个函数中自动地决定一个数组编译时的大小。

指向数组元素的指针参数在文本处理中非常普遍。函数 (`str_cpy`) 就可以把一个字符串拷贝到另一个字符串中 (除了无返回值外, 它与 `strcpy` 一样):

```

void str_cpy(char *s1, const char *s2)
{
    while (*s1++ == *s2++);
}

```

`while` 循环测试不是为了检测值相等, 而是检测在赋值后 (但是在自增之前) `s1` 的值, 循环在拷贝了结束符号 '`\0`' 后停止, 表达式 `*p++` 是一个很常用的 C/C++ 习惯方式。

习题 2.2

下面的语句通过一系列指针表达式修改字符串 `s`, 当顺序执行时每个表达式重新得到的是什么字符, 最后的结果是什么?

```

char s[ ] = " desolate", *p = s;

*p++ == ?
*(p++) == ?
(*p)++ == ?
*++p == ?
*(++p) == ?
++*p == ?

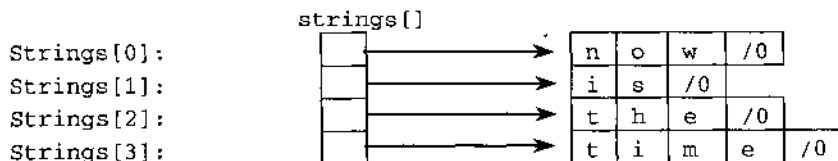
++(*p) == ?
Strcmp(s, ?) == 0

```

(感谢 Lincoln 实验室的 Chet Small 提供了这个非常聪明的例子)。

2.9 字符串数组

有两种方式来描述 C 风格的字符串数组：(1) 指针数组；(2) 二维数组。程序清单 2.13 中的程序说明了第一种方式。内存分布如下：



程序清单 2.13 用指向字符的指针数组来实现字符串

```
// array6.cpp:粗糙的数组
#include <iostream>
#include <cstring>
using namespace std;

main()
{
    char* strings[] = {"now", "is", "the", "time"};
    size_t n = sizeof strings / sizeof strings[0];

    //从粗糙的数组打印
    for (int i = 0; i < n; ++i)
        cout << "String " << i << " == \" " << strings[i]
              << "\", \tsize == " << sizeof strings[i]
              << ", \tlength == " << strlen(strings[i])
              << endl;
}

//输出:
String 0 == "now",    size == 4,    length == 3
String 1 == "is",    size == 4,    length == 2
String 2 == "the",    size == 4,    length == 3
String 3 == "time",    size == 4,    length == 4
```

由于字符串可以有不同的长度，所以这一类型的数组有时被称为粗糙的 (ragged) 数组。这一方式仅使用了容纳数据所需的内存数量，再加上指向每个字符串的指针。运行时系统传递给 main 函数的命令行参数数组 argv 是一个粗糙的数组。

粗糙的数组方式的一个不利之处是，在大多数环境中，需要动态地为每个字符串分配内存 (参见第 20 章)。如果你不介意浪费一小部分空间，而且如果你也知道会遇到的最长的字符串的长度，就可以使用一个固定大小的区域来存储二维字符数组 (每行一个字符串)。程序

清单 2.14 中数组的内存区域分布如下：

	array[][5]				
array[0]:	n	o	w	/0	?
array[1]:	i	s	/0	?	?
Array[2]:	t	h	e	/0	?
Array[3]:	t	i	m	e	/0

程序清单 2.14 作为二维字符数组中的行来实现字符串

```
// array7.cpp: 在二维字符数组中存储字符串
#include <iostream>
#include <cstring>
using namespace std;

main()
{
    char array[][5] = {"now", "is", "the", "time"};
    size_t n = sizeof array / sizeof array[0];

    for (int i = 0; i < n; ++i)
        cout << "array[" << i << "] == \"< < array[i]
            << "\", \tsize == " << sizeof array[i]
            << ", \tlength == " << strlen(array[i])
            << endl;
}

//输出:
array[0] == "now",    size == 5,    length == 3
array[1] == "is",     size == 5,    length == 2
array[2] == "the",    size == 5,    length == 3
array[3] == "time",   size == 5,    length == 4
```

正如这个程序所表明的，如果多维数组的第一维能从它的初始化中推断出来，那么就不必具体指出多维数组的第一维。

在程序设计语言中 C++ 在某种程度上是独一无二的，因为在使用数组时仅可以使用其部分下标。就像程序清单 2.14 中的程序所使用的那样，表达式 `array[i]` 是指向第 `i` 行的指针。对于一个定义为 `int a[2][3][4]` 的数组，`a[i]` 代表的是什么呢？而 `a[i][j]` 又是什么呢？继续读下一节。

2.10 指针和多维数组

实际上，在 C++ 中没有多维数组！至少对多维数组没有直接的支持。人们通常把一个一维数组看作一个向量，把一个二维数组看作一个表或者矩阵，把一个三维数组看作一个长方体。然而，数组的几何模型使明智地使用高维数组变得很困难，取而代之的是 C++ 支持“数

组的数组”的概念。例如，如果一个一维的整型数组为

```
int a[4]={0,1,2,3};
```

它是一个有索引的整数集合：

0	1	2	3
---	---	---	---

我们通常把它描述成一个向量：

0	1	2	3
---	---	---	---

对一个二维整型数组，如：

```
int a[3][4]={{0,1,2,3},{4,5,6,7},{8,9,0,1}}
```

是像下面这样的向量集合：

0	1	2	3	4	5	6	7	8	9	0	1
a[0]				a[1]				a[2]			

或者，如果喜欢的话可写成：

a[0]	0	1	2	3
a[1]	4	5	6	7
a[2]	8	9	0	1

这就是数组的集合。因此，a 就是一个“3 个具有 4 个整型元素的数组的数组”，a[0] 就是这些具有 4 个整型元素的数组之一。由于在表达式中使用数组名时，它总是被解释为指向它的第一个元素的指针，所以一个诸如 a+1 之类的表达式就是“指向一个具有 4 个整型元素的数组的指针”，在这种情况下，该表达式将指向第二行（即 a[1]）。程序清单 2.15 中的程序说明了如何声明指向一个数组的指针，这样的指针可以在不改变下标语法的情况下，替换数组名。初学者容易错误地假设：指向整型数据的指针能代替一个整型数组名，就像以下程序：

```
int a[]={0,1,2,3},*p=a;
/*...*/
p[i]=...
```

那么，这样指向整型数据的指针的指针将会对二维数组进行同样的操作，就像下面：

```
int a[][4]={ {0,2,3,4},{4,5,6,7},{8,9,0,1}};
int **p=a;      /*受怀疑的指针转换*/
/*...*/
p[i][j]=...
```

要弄清以上做法为何不对，考虑一下表达式 p[i][j]，根据“重要的指针原则 2”，这就相当于：

```
*(p[i]+j)
```

也相当于：

```
*(*(p+i)+j)
```

程序清单 2.15 说明在二维数组中指向一维数组的指针

```
// array8.cpp: 使用一个一维数组指针
#include <iostream>
```

```

main()
{
    int a[][4] = {{0,1,2,3},{4,5,6,7},{8,9,0,1}};
    int (*p)[4] = a;          //指向包含 4 个整型成员的数组
    size_t nrows = sizeof a / sizeof a[0];
    size_t ncols = sizeof a[0] / sizeof a[0][0];

    cout << "sizeof(*p) == " << sizeof *p << endl;
    for (int i = 0; i < nrows; ++i)
    {
        for (int j = 0; j < ncols; ++j)
            cout << p[i][j] << ' ';
        cout << endl;
    }
}

//输出:
sizeof(*p) == 16
0 1 2 3
4 5 6 7
8 9 0 1

```

由于 p 是一个指向整型数据的指针，所以表达式 $p+i$ 往 p 后面移动的距离等于 i 个指针的大小，而不是 i 行（我们需要移动 i 行）。显然，我们需要的指针类型是 p 所指向的指针具备一个行的大小，因此 p 必须是一个行指针，也就是说它指向一个大小合适的数组。因此，有趣但又有逻辑性的语法为：

```
int (*p)[4]=a;
```

根据 p 的这个定义，编译器依据下面的步骤来求表达式 $*(* (p+i) + j)$ 的值：

1. $p+i$

这一步计算行指针，该行超出行 p 当前所指向的 i 行。

2. $* (p+i)$

这是具体的行（是一个数组）。

3. 由于数组 $* (p+i)$ 不是 `sizeof` 或者 `&` 运算的操作数，所以它被指向它第一个元素的指针所替代，即 $\&a[i][0]$ ，这是一个指向整型（`int`）的指针。

4. $\&a[i][0]+j$

因为 $\&a[i][0]$ 是一个指向整型（`int`）的指针，加上 j 就把这个指针向前移动了 j 个整型数据单位，结果是 $\&a[i][j]$ 。

5. $*\&a[i][j]$

这就是整型 $a[i][j]$ 。

表 2.1 总结了指针和二维数组的这种关系，注意，不要因为 $a+1$ 和 $a[1]$ 有同样的值（ $0 \times$

8) 而得出以下的结论:

```
a[i]==a+i // 错误!
```

它们仅仅在数值上相等, 而类型却不一样, 因为 `sizeof (a+1) == 2` (一个指针), 而 `sizeof (a[1]) == 8` (一个有 4 个整型数的数组)。

2.11 更高深的内容

我们可以很自然地得出以下结论: 一个三维数组是二维数组的集合。

```
int a[2][3][4] = {{{0,1,2,3},{4,5,6,7},{8,9,0,1}},
                  {{2,3,4,5},{6,7,8,9},{0,1,2,3}}};
```

0	1	2	3
4	5	6	7
8	9	0	1

a[0]

2	3	4	5
6	7	8	9
0	1	2	3

a[1]

这个数组的第一个元素是一个“二维数组” `a[0]` (从技术上来说, `a[0]` 是一个由 3 个含有 4 个整数的数组的数组), 为了使指针与数组名 `a` 一致, 定义:

```
int (*p)[3][4] = a;
```

程序清单 2.16 是一个应用该指针的程序示例, 表 2.2 是与表 2.1 相似的三维数组。

表 2.1 对于二维数组的数组指针转换

表达式	类 型	值
<code>a</code>	由 3 个含有 4 个整型 (int) 元素的数组组成的二维数组	0x0
<code>a+1</code>	指向具有 4 个整型 (int) 数组的指针	0x10
<code>a[1]</code>	具有 4 个整型 (int) 元素的数组	0x10
<code>a[1]+1</code>	指向整型 (int) 的指针	0x14
<code>a[1][1]</code>	整型 (int)	5

其中十六进制值是相对于 `a` 的地址偏移量。

程序清单 2.16 说明在三维数组内指向二维数组的指针

```
// array9.c: 使用一个二维数组指针
#include <iostream>
using namespace std;

main()
{
    int a[][3][4] = {{{0,1,2,3},{4,5,6,7},{8,9,0,1}},
                     {{2,3,4,5},{6,7,8,9},{0,1,2,3}}};
```

```

int (*p)[3][4] = a;
size_t ntables = sizeof a / sizeof a[0];
size_t nrows  = sizeof a[0] / sizeof a[0][0];
size_t ncols  = sizeof a[0][0] / sizeof a[0][0][0];

cout << "sizeof(*p) == " << sizeof *p << endl;
cout << "sizeof(a[0][0]) == " << sizeof a[0][0] << endl;
for (int i = 0; i < ntables; ++i)
{
    for (int j = 0; j < nrows; ++j)
    {
        for (int k = 0; k < ncols; ++k)
            cout << p[i][j][k] << ' ';
        cout << endl;
    }
    cout << endl;
}

//输出:
sizeof(*p) == 48
sizeof(a[0][0]) == 16
0 1 2 3
4 5 6 7
8 9 0 1

2 3 4 5
6 7 8 9
0 1 2 3

```

表 2.2 关于三维数组的数组指针转换

表 达 式	类 型	值
a	2 个具有 3 个含有 4 个整型 (int) 元素数组的数组组成的数组	0x0
a+1	指向由 3 个有 4 个整型 (int) 元素的数组的数组	0x30
a[1]	由 3 个有 4 个整型 (int) 元素的数组组成的数组	0x30
a[1]+1	指向有 4 个整型 (int) 元素的数组的指针	0x40
a[1][1]	有 4 个整型 (int) 的数组	0x40
a[1][1]+1	指向整型 (int) 的指针	0x44
a[1][1][1]+1	整型 (int)	7

其中，十六进制值是相对于 a 的地址偏移量。

在程序清单 2.15 和程序清单 2.16 中的程序表明如何确定一个数组和它所有的子数组的秩

(即维数), 对于程序清单 2.15 中的二维数组 a , 秩是它所包含的行数 (一维对象), 也就是:

```
sizeof a /sizeof a[0]
```

从名称上讲, 每一个行的秩就是 a 中每一行的整数的个数 (0 维对象):

```
sizeof a[0] /sizeof a[0][0]
```

总的来说, 如果 a 是一个 n 维数组, 那末, a 为 $n-1$ 维对象

```
sizeof a /sizeof a[0]
```

的集合, a 中每一个 $(n-1)$ 维对象包含 $n-2$ 维对象

```
sizeof a[0] /sizeof a[0][0]
```

每一个 $(n-2)$ 维对象依次包含 $(n-3)$ 维对象

```
sizeof a[0][0] /sizeof a[0][0][0]
```

等等, 直到每一个一维对象中零维对象的个数为

```
sizeof a[0][0]...[0] /sizeof a[0][0]...[0][0]
      {n-1 subscripts}      {n-1 subscripts}
```

练习 2.3

模仿表 2.1 和表 2.2, 完成下面四维数组的数组指针转换表:

```
int a[2][3][4][5] =
{
  {
    { {0,1,2,3,4},{5,6,7,8,9},{0,1,2,3,4},{5,6,7,8,9} },
    { {0,1,2,3,4},{5,6,7,8,9},{0,1,2,3,4},{5,6,7,8,9} },
    { {0,1,2,3,4},{5,6,7,8,9},{0,1,2,3,4},{5,6,7,8,9} }
  },
  {
    { {0,1,2,3,4},{5,6,7,8,9},{0,1,2,3,4},{5,6,7,8,9} },
    { {0,1,2,3,4},{5,6,7,8,9},{0,1,2,3,4},{5,6,7,8,9} },
    { {0,1,2,3,4},{5,6,7,8,9},{0,1,2,3,4},{5,6,7,8,9} }
  }
};
```

0	1	2	3	4
5	6	7	8	9
0	1	2	3	4
5	6	7	8	9

0	1	2	3	4
5	6	7	8	9
0	1	2	3	4
5	6	7	8	9

0	1	2	3	4
5	6	7	8	9
0	1	2	3	4
5	6	7	8	9

== a[0]

0	1	2	3	4
5	6	7	8	9
0	1	2	3	4
5	6	7	8	9

0	1	2	3	4
5	6	7	8	9
0	1	2	3	4
5	6	7	8	9

0	1	2	3	4
5	6	7	8	9
0	1	2	3	4
5	6	7	8	9

== a[1]

表 达 式	类 型	值
a		
a+1		
a[1]		
a[1]+1		
a[1][1]		
a[1][1]+1		
a[1][1][1]		
a[1][1][1]+1		
a[1][1][1][1]		

2.12 指向函数的指针

一个指针可以指向函数也可以指向存储的对象。下面的语句声明 **fp** 是一个指向返回值为整型 (int) 的函数的指针:

```
int (*fp) ( );
*fp 的圆括号是必需的, 没有它的语句
int *fp ( );
```

将 **fp** 声明为一个返回指向整型 (int) 指针的函数。这就是将星号与类型声明紧密相连的方式成为逐渐受人们欢迎的方式的原因之一。

```
int* fp(); //方式说明 fp() 返回一个指向整型的指针(int *)
```

当然, 这种方式建议你通常应该每条语句只声明一个实体, 否则, 以下的语句将会使人感到迷惑:

```
int *ip, jp; //jp 不是一个指针!
```

如果想具体说明 **fp** 指向的函数就必须带有一定的参数, 如果是一个浮点型 (float) 和一个字符串型, 那么可以这样写:

```
int (*fp) (float, char * );
```

然后可以在 **fp** 中存储这样一个函数的地址:

```
extern int g(float, char*);
```

```
fp=g;
```

表达式中函数的名字解析地址可以认为是指向那个函数代码区的开始的地址。下面的“hello, world”程序说明如何通过指针来执行一个函数。

```
/*hello.c: 通过函数的指针来说 hello */
#include <stdio.h>
main()
{
```

```
void ( * fp )()=printf;
fp("hello, world\n");
}
```

要通过指针来执行一个函数，你可能认为得这样写：

```
(*fp)("hello world\n");
```

来复引用指针。事实上，在 ANSI C 出现以前必须这样做，但是 ANSI C 委员会决定容许像我在 `hello.c` 中那样使用的普通函数调用句法。由于编译器知道它是一个指向函数的指针，并且它还知道在该环境下所能做的惟一的一件事就是调用函数，因此这里没有任何模糊不清的表达。

当把函数名作为一个参数传递给另一个函数时，编译器实际上给这个函数传递了一个指针（与数组名类似）。但是你为什么曾经想过给另一个函数传递函数指针呢？C 标准库中的排序函数 `qsort` 的使用就是一个例子，采用简单和复合的排序关键字，它可以对由任何类型的元素所组成的数组排序。程序清单 2.17 中的程序说明怎样排序命令行参数字符串，在这种情况下所需要做的全部事情就是传递给 `qsort` 一个知道如何比较字符串的函数。一个函数（像 `qsort`）通过由运行时决定的指针来调用另一个函数（像 `comp`）的行为叫做返调（callback）。

函数指针的数组在菜单驱动应用程序中很容易见到，假设想将以下的目录展现给用户：

1. 返回
2. 插入
3. 更新
4. 退出

程序清单 2.18 的程序直接把键盘输入作为索引放到指向处理每一个菜单选择函数的指针的数组中。

程序清单 2.17 用 `qsort` 函数将命令行参数排序

```
// sortargs.cpp: 排序命令行参数
#include <iostream>
#include <cstring>
#include <cstdlib>
using namespace std;

int comp(const void*, const void*);

main(int argc, char *argv[])
{
    qsort(argv+1, argc-1, sizeof argv[0], comp);
    while (--argc)
        cout << *++argv << endl;
}

int comp(const void* p1, const void* p2)
```

```
{
    const char *ps1 = * (const char**) p1;
    const char *ps2 = * (char**) p2;
    return strcmp(ps1,ps2);
}
```

//从"sortargs *.cpp"命令输出:

```
address.cpp
arith.cpp
array1.cpp
array2.cpp
array3.cpp
array4.cpp
array5.cpp
array6.cpp
array7.cpp
array8.cpp
array9.cpp
bit1.cpp
bit2.cpp
convert.cpp
inspect.cpp
pointer.cpp
ptr2ptr.cpp
sortargs.cpp
swap1.cpp
swap2.cpp
```

程序清单 2.18 用函数指针数组来处理菜单选择

```
/* menu.c: 举例说明函数数组 */
#include <stdio.h>

/*你一定要给这些提供定义*/
extern void retrieve(void);
extern void insert(void);
extern void update(void);
extern int show_menu(void);          /*返回 keypress */

main()
{
    int choice;
    void (*farray[])(void) = {retrieve,insert,update};

    for (;;)

```

```

    {
        choice = show_menu();
        if (choice >= 1 && choice <= 3)
            farray[choice-1]();          /* 进程的需要 */
        else if (choice == 4)
            break;
    }
    return 0;
}

```

2.13 指向成员函数的指针

如果回调函数是某个类的成员函数将会怎样？获得指向类成员的指针与获得指向非成员实体的指针的方式相似，只存在很小的语法变化。例如，考虑下面类的定义：

```

class C
{
public:
    void f ( ) {cout << "C::f\n";}
    void g ( ) {cout << "C::g\n";}
};

```

可以这样定义一个指向 C 类成员函数的指针：

```

void (C::*pmf) ( ); // pmf 是指向 C 的成员函数的指针
                    // 不带参数并且没有返回值

```

可以根据需要来初始化 pmf 使其指向不同的成员函数，如下面的程序所示：

```

main ( )
{
    C c;
    void (C::*pmf) ( ) =&C::f;
    (c.*pmf) ( );
    pmf =&C::g;
    (c.*pmf) ( );
}
// 输出：
C::f
C::g

```

· * 运算符调用了代表其左边操作数所指向的对象的成员函数。如果 cp 是一个指向 C 的指针，那么可以使用->*运算符，就像：

```

( cp->*pmf ) ( );

```

由于 C++ 定义运算符优先级的方式，所以圆括号是完全确认所调用的函数所必需的。请参见程序清单 2.19（程序清单 2.18 菜单例子的指向成员函数指针的版本）。

程序清单 2.19 举例说明对成员的指针

```
// menu2.cpp: 使用指向函数成员的指针
#include <iostream>
using namespace std;

class Object
{
public:
    void retrieve() {cout << "Object::retrieve\n";}
    void insert() {cout << "Object::insert\n";}
    void update() {cout << "Object::update\n";}
    void process(int choice);

private:
    typedef void (Object::*Omf)();
    static Omf farray[3];
};

Object::Omf Object::farray[3] =
{
    &Object::retrieve,
    &Object::insert,
    &Object::update
};

void Object::process(int choice)
{
    if (0 <= choice && choice <= 2)
        (this->*farray[choice])();
}

main()
{
    int show_menu();    //你所提供的!
    Object o;

    for (;;)
    {
        int choice = show_menu();
        if (1 <= choice && choice <= 3)
            o.process(choice-1);
        else if (choice == 4)
            break;
    }
}
```


2.14 封装和不完全类型

好的编程习惯能隐藏用户不必知道的执行细节。例如，要执行一个整数栈，可以为用户提供如下的头文件：

```
// mystack.h
class StackOfInt ( )
{
public:
    StackOfInt ( ) ;
    void push ( int );
    int pop ( ) ;
    int top ( ) const;
    int size ( ) const;
private:
    enum { MAX_STACK = 100 };
    int data [ MAX_STACK];
    int stkPtr;
};
```

因为下面几行的数组和栈指针是私有的，因此用户必须通过你提供的公共接口按照你的方式 进行操作（程序清单 2.20 就是一个例子）。即使成员函数的实现对于用户是隐藏的，但他（或她）可以看一下头文件就很容易地推断出与 程序清单 2.21 相似的内容。如果隐藏所有的实现细节对你来说很重要，你可以通过使用不完全类型增加一个额外的保护层。

不完全类型的大小不能在编译期的时候确定。下面的声明就是一个例子：

```
extern int a [];
```

这个声明定义 `a` 是一个未知长度的数组，所以不能使用 `sizeof`，否则将得到一个错误信息。这个定义完全不同于：

```
extern int *a;
```

这是一个有大小的指针。可以通过在另一个类中隐藏它的实现细节来加强上面栈类型的封装性。在程序清单 2.22 中，只有一个指向 `StackImp` 的指针出现在 `StackOfInt` 的私有部分中。语句

```
class StackImp;
```

是 `StackImp` 的一个不完全声明，它仅仅表明了类的存在。一旦只有一个 `StackImp` 的指针或引用出现在 `stack2.h` 中，用户就不需要 `stkimp.h`。 `StackOfInt` 的成员函数目前只是传递请求到 `StackImp`（见程序清单 2.23 和程序清单 2.24）。

程序清单 2.20 StackOfInt 的定义

```
// tstack.cpp: 使用 StackOfInt 类
#include <iostream>
#include "mystack.h"
```

```
using namespace std;

main()
{
    StackOfInt s;

    s.push(10);
    s.push(20);
    s.push(30);
    while (s.size())
        cout << s.pop() << endl;
}

//输出:
30
20
10
```

程序清单 2.21 StackOfInt 的实现

```
// stack.cpp: StackOfInt 类的实现
#include "stack.h"

StackOfInt::StackOfInt()
{
    stkPtr = 0;
}

void StackOfInt::push(int i)
{
    if (stkPtr == MAX_STACK)
        throw "overflow";
    data[stkPtr++] = i;
}

int StackOfInt::pop()
{
    if (stkPtr == 0)
        throw "underflow";
    return data[--stkPtr];
}

int StackOfInt::top() const
{
    if (stkPtr == 0)
        throw "underflow";
```

```

        return data[stkPtr - 1];
    }

    int StackOfInt::size() const
    {
        return stkPtr;
    }

```

程序清单 2.22 为了更好地封装而使用不完全类型

```

// stack2.h: 隐藏堆栈的实现
class StackImp;

class StackOfInt
{
public:
    StackOfInt();
    ~StackOfInt();
    void push(int);
    int pop();
    int top() const;
    int size() const;

private:
    StackImp* imp;
};

```

程序清单 2.23 StackOfInt 类的实现

```

// stack2.cpp
#include "stack2.h"
#include "stkimp.h"

StackOfInt::StackOfInt()
    : imp(new StackImp)
{

}

StackOfInt::~StackOfInt()
{
    delete imp;
}

void StackOfInt::push(int i)
{

```

```
        imp->push(i);
    }

    int StackOfInt::pop()
    {
        return imp->pop();
    }

    int StackOfInt::top() const
    {
        return imp->top();
    }

    int StackOfInt::size() const
    {
        return imp->size();
    }
}
```

程序清单 2.24 堆栈的实现

```
// stkimp.cpp
#include "stkimp.h"

StackImp::StackImp()
{
    stkPtr = 0;
}

void StackImp::push(int i)
{
    if (stkPtr == MAX_STACK)
        throw "overflow";
    data[stkPtr++] = i;
}

int StackImp::pop()
{
    if (stkPtr == 0)
        throw "underflow";
    return data[--stkPtr];
}

int StackImp::top() const
{
    if (stkPtr == 0)
```

```

        throw "underflow";
    return data[stkPtr - 1];
}

int StackImp::size() const
{
    return stkPtr;
}

```

2.15 小结

- C 和 C++ 仅仅与那些使用它们的人一样危险。
- 指针是地址。
- 可以将任何一个指针赋值成 `void*`。
- 注意区分一个 `const` 指针和一个指向 `const` 的指针。
- $p \pm n == (\text{char}^*) p \pm n * \text{sizeof}(*p)$ 。
- $p - q == \pm n$ 。
- $*(a+i) == a[i]$ 。
- 除非在 `sizeof` 和 `&` 的上下文中，否则一个数组名即是指向它第一个元素的指针；
- 没有多维数组，只有数组的数组。
- 仅是指针的存在并不要求它所引用的类型的实现的有效性（这是一个不完全类型）。

如果理解了这些概念，你就正在逐渐地成为一名可信赖的 C++ 程序员。现在去告诉你的老板，她可以指派你去编写真正的程序。

练习答案

练习 2.1

已知如下声明：

```
int a[ ] = { 10, 15, 4, 25, 3, -4 };
int *p = &a[ 2 ];
```

下面表达式的结果是什么？

- $*(p+1)$ 25
- $p[-1]$ 15
- $p - a$ 2
- $a[*p++]$ 3
- $*(a+a[2])$ 3

练习 2.2

下面的语句通过一系列指针表达式修改字符串 `s`，当顺序执行时每个表达式重新得到什

么字符, 最后的结果是什么?

```
char s[ ] = "desolate", *p = s;
*p++ == d
*(p++) == e
(*p)++ == s
*++p == o
*(++p) == l
++*p == m
++(*p) == n
strcmp(s, "detonate") == 0
```

练习 2.3

表 达 式	类 型	值
a	2 个有 3 个含有 4 个由 5 个整型数组所组成的数组所组成的数组的数组	0x0
a+1	指向 3 个由包含 4 个由包含 5 个整型数组的数组所组成的数组	0x0f0
a[1]	3 个由包含 4 个由包含 5 个整型数组的数组所组成的数组	0x0f0
a[1]+1	指向 4 个由包含 5 个整型数的数组所组成的数组	0x140
a[1][1]	4 个由包含 5 个整型数的数组所组成的数组	0x140
a[1][1]+1	指向由个整型所组成的数组的指针	0x154
a[1][1][1]	由 5 个整型所组成的数组	0x154
a[1][1][1]+1	指向整型的指针	0x158
a[1][1][1][1]	整型	6

其中十六进制值是相对于 a 地址偏移量。

0x0f0 == 240

0x140 == 320

0x154 == 340

0x158 == 344

预处理器

编译器在开始通常的语法检查和指令解释之前，它会让程序进入一个称为预处理的初始状态，预处理根据你的指令更改编译器所看见的程序的文本。那个编译器所见的，并被更改过的文本称为翻译单元（translation unit）。特别地，预处理实现下面三个功能：

1. 头文件或源文件包含；
2. 宏扩展；
3. 条件编辑。

3.1 #include 指令

任何一个 C 程序员看到或写的第一行源代码是：

```
#include <stdio.h>
```

如果觉得有挑战性，你不妨马上花点时间写出自己所知道的关于这条语句的一切。

让我们看看你是如何做的。当然 `stdio.h` 是一个标准库的头文件，之所以这么说是因为这样的指令通常出现在源文件开始的附近，因而它们的定义会被强制存在于编译的其余部分。我们一般认为它是头文件，但是在标准库中这些定义和声明不必嵌入文件中。在编译之前 C 和 C++ 标准只要求将所需要的定义替代程序文本包含的指令。它们可以驻留在预处理器内部的表中。例如，大部分 PC 编译器在适当的子目录中安装头文件。但是，实现既不必在物理文件中提供头文件信息，也不必在指令之后指明这些文件。那么编译器怎样才能在一个其文件系统不允许在文件名中出现句点的平台上提供名为 `stdio.h` 的文件呢？

与其他编译器相似，该编译器也支持用引号限定的包含指令：

```
#include "mydefs.h"
```

字符串必须描述一个能被本地文件系统识别的名字，相应的文件必须是一个有效的

C/C++源文件，就像标准头文件一样，应该包含类的声明、函数原型、宏定义和其他声明。实现必须提供它用来查找所需源文件的文件机制。在具备层次化文件系统的平台上，大多数编译器首先查找当前目录，失败后再试着查找标准头文件的所在目录。因为标准头文件的名字是特殊预处理记号，而不是字符串，所以在头文件名中出现的任何反斜杠符都不是转义字符。

```
#include <sys\stat.h>          /*是\，不是\\*/
#include "\project\include\ mydefs.h"    /*同上*/
```

被包含的文件中可以包含其他包含语句，嵌套在实现定义的限定中（检查你的文件）。由于一些定义，如 `typedefs`，在一次编译中必须只出现一次，你就一定要警惕有可能一个文件被重复包含。常用的技巧是定义一个与文件相关的符号，如果编译器已经见过这个符号，那么在编译时就排除这个文件的文本：

```
// mydefs.h
#ifndef MYDEFS_H
#define MYDEFS_H
<declarations/definitions go here>
#endif
```

3.2 其他的预处理指令

除了 `#include` 指令外还有其他许多预处理指令：还有 11 种其他预处理指令可以用有意义的方式来改变源代码文本（见表 3.1）。所有的指令都以 `#` 符号开始，它必须是源代码行中的第一个非空格字符。

表 3.1 预处理指令

指 令	描 述
<code>#include</code>	包含头文件或源代码文件的文本
<code>#define</code>	为当前编译单元用一个可选的值加入一个符号到符号表中
<code>#undef</code>	从符号表中删除符号
<code>#if</code>	条件编译控制流指令
<code>#elif</code>	
<code>#else</code>	
<code>#endif</code>	
<code>#ifdef</code>	符号表查询指令（也用于条件编译）
<code>#ifndef</code>	输出信息到标准错误输出并终止程序
<code>#error</code>	重新计算当前的代码行。其作用就像代码生成器是用来同步化在错误消息中用原始代码行产生的行
<code>#line</code>	
<code>#pragma</code>	与编译器有关的行为

`#define` 指令生成宏定义。宏是一系列 0 个或多个预处理记号的名字。有效的预处理记号包括有效的 C++ 语言记号，如标识符、字符串、数字、运算符、预处理指令、头文件名和任何单一的字符。例如，代码行。

```
#define MAXLINES 500
```

使文本“500”（无引号）与符号 `MAXLINES` 相关联。预处理器保存着一个由 `#define` 指令产生的所有符号的表和相应的替换文本。无论何时当预处理器在注释和带引号的字符串以外遇到 `MAXLINES` 符号时，将用 500 代替它。重要的是，应该记住这是文本替换，在编译后期的阶段它将在编译器中出现，就好像实际输入 500 来替代 `MAXLINES` 一样。在预处理期间没有语义分析。

无参数宏就像上面的 `MAXLINES`，有时被称为类对象宏，因为它定义一个像对象一样的程序常数。因为类对象宏类似常数，习惯上将其写成大写字母以提示读者，然而，很少有必要定义类对象宏，应该用 `const` 代替变量。语句

```
const int MAXLINES = 500;
```

比下边的宏定义有很多优越性：

```
#define MAXLINES 500
```

由于编译器知道对象的语义，你可以进行更强大的编译期类型检查。也可以用符号调试器来检查 `const` 对象。全局 `const` 对象如果不显式地声明是全局的，将作为内部的链接，因此可以安全地用 `const` 定义替换所有的类对象宏。

也可以定义带有零或多个参数的类函数宏，如：

```
#define beep() putc('\a',stderr)
#define abs(x) ((x)>=0 ? (x) : ((-x)))
#define max(x, y) (((x)>(y)) ? (x) : (y))
```

在宏名和左边第一个括号之间一定不能有空格。表达式

```
abs (-4 )
```

展开为：

```
((-4)>=0 ? (-4) : (-(-4)))
```

就好像是这样输入表达式一样。在替换文本中把宏参数（像上面的 `x`）用括号括起来是很重要的，这将避免由复杂的参数表达式所引起的优先权问题。例如，如果幼稚地用数学表达式定义一个绝对值但没加括号：

```
x >= 0 ? x : -x
```

那么，表达式 `abs(a-1)` 将展开为：

```
a - 1 >= 0 ? a - 1 : - a - 1
```

当 `a-1 < 0` 时将出错（应该是 `-(a-1)`）。

即使在所有变量外都加了括号，也还需要在整个要替换的表达式外加上括号，以避免周围文本的干扰。为了解释这点，定义 `abs` 没有外层括号：

```
#define abs (x) (x)>=0 ? (x) : (-x)
```

则 `abs(a)-1` 将展开为：

```
((a)>=0 ? (a) : -(a))-1
```

而当 a 为非负数时上式就错了 (-1 丢失了)。

使用有其他影响的表达式作为宏参数也是危险的, 例如, 宏调用 `abs(i++)` 展开为:

```
( ( i ++ ) >= 0 ? ( i ++ ) : ( - ( i ++ ) ) )
```

无论 i 是什么值都将增加两次而不是一次, 这也许并不是你所希望的。

类函数的宏在 C++ 中几乎是没有什么必要的, 可以用内联函数替换大多数的类函数的宏。例如, 可以用下边的函数替代上面的 `max` 宏。

```
inline int max( int x, int y)
{
    return x >= y ? x : y;
}
```

不必加括号来避免优先权争议。因为这是一个真正的有作用域和类型检查的函数。也不必担心像使用宏那样有其他影响, 如在以下调用中:

```
max( x++, y++)
```

但是宏的形式具有能接受任何类型的参数的优势, 你说呢? 没问题。可以将 `max` 定义为模板来替换:

```
template<class T>
inline int max( int x, int y)
{
    return x >= y ? x : y;
}
```

这个函数模板和其他许多的模板一起出现在标准头文件 `<algorithm>` 中。

一定要记住的是内联 (`inline`) 只是对编译器的一个提示, 并非所有的函数都服从内联, 尤其那些带循环和复杂的控制结构的函数。当编译器无法内联函数时, 它将会告诉你。在大多数情况下, 定义一个函数比定义一个宏而丧失真正函数提供的类型安全要好。

3.3 预定义宏

相容实现提供如表 3.2 所示的系统预定义的实现对象的宏。最后三个在源文件的编辑期间保持常数。编译器提供的任何其他预定义的宏必须以下划线开头, 其后跟着大写字母或另一个下划线。C 翻译器也定义 `_STDC_`, 它在标准模式下编译时为非 0。大多数编译器都提供多个编译模式, 其中一些不是标准的。既不能用 `#define` 指令对这些宏进行重定义, 也不能用 `#undef` 指令迁移它们。相容程序清单 3.1 在一个样本平台上说明了这些宏。

表 3.2

预定义宏

宏	值
<code>__LINE__</code>	当前代码行的行号 (等于读到目前为止新行的字符数)
<code>__FILE__</code>	源文件的名字
<code>__DATE__</code>	以 “Mmm dd yyyy” 的形式转换日期

续表

宏	值
<code>__TIME__</code>	以“hh:mm:ss”的形式转换时间
<code>__cplusplus</code>	一个包含正式的 C++ 标准的日期

一致编译器也提供类函数的宏——`assert`，可以用它在程序中加入诊断。如果它的参数值为 0，`assert` 将把参数连同源文件和代码行（用 `__FILE__` 和 `__LINE__`）一块打印出来，并中断程序（见程序清单 3.2）。

程序清单 3.1 输出预定义宏

```
// sysmac.cpp: 打印系统宏
#include <iostream>
using namespace std;

main()
{
    cout << "__DATE__ == " << __DATE__ << endl;
    cout << "__FILE__ == " << __FILE__ << endl;
    cout << "__LINE__ == " << __LINE__ << endl;
    cout << "__TIME__ == " << __TIME__ << endl;
}
```

//输出:

```
__DATE__ == Nov 28 1996
__FILE__ == sysmac.cpp
__LINE__ == 7
__TIME__ == 09:37:38
```

程序清单 3.2 举例说明判断失败

```
// fail.cpp
#include <iostream>
#include <cassert>
using namespace std;

main()
{
    int i = 0;
    assert(i > 0);
}
```

```
//输出:
```

```
C:>fail
```

```
Assertion failed: i > 0, file fail.cpp, line 8
```

```
Abnormal program termination
```

C 编译器被允许为标准 C 库中的任何函数提供宏形式（如 `getc` 和 `putc` 为了效率经常被当作宏来用）。除了少数要求用类函数的宏（如 `assert`、`setjmp`、`va_arg`、`va_end` 和 `va_start`）以外，实现必须为 C 标准库中的所有函数提供真实的函数形式。库函数中的宏形式实际上在编译时隐藏其原型，所以其参数在翻译时不进行类型检查。为了加强调用真正的函数，用 `#undef` 指令去掉宏定义。例如：

```
#undef getc
```

或者，当调用它时可以把函数名用括号括起来：

```
C = (getc)(stdin);
```

这与宏定义不相符，因为左边的括号没有紧跟函数名。无论标准 C++ 库中的全局函数是否被定义为内联形式，其都是由实现来定义的。

3.4 条件编译

可以用条件指令有选择地包含或拒绝代码段。当想注释大段代码时，`#if` 指令是很方便的。不能只用简单的注释来隐藏这个部分，因为代码本身中包含了 C 风格的注释，这会导致外部的注释提前终止。最好是使代码在 `#if` 指令总是为 0 的询问语句中：

```
#if 0
```

```
<将要忽略的代码放在此处>
```

```
#endif
```

`#if` 指令的目标必须为整型常量，并且符合 C 的一般规则即非 0 为真，0 为假。在这种表达式中不可以采用强制类型转换或 `sizeof` 运算符。

如果代码是在 C/C++ 混合编程环境中编写的，可以将新的 C++ 代码与旧的 C 代码相链接。所需要做的所有事情就是通过 `extern "C"` 链接规范来告诉 C++ 翻译器，不要“砍掉”与 C 组件相匹配的外部名字，例如：

```
extern "C" void f();          //f() 在 C 环境下被编译
```

C++ 标准库中的 C 部分需要知道是否该砍掉名字（在第 1 章已解释过），这取决于在什么模式下（C 或 C++）编译。如果仔细阅读标准 C 头文件会发现销售商使用 `__cplusplus` 宏通过一个 `extern "C"` 块根据以下模式条件隐藏了标准 C 声明：

```
#if defined(__cplusplus)
```

```
extern "C"
```

```
{
```

```
#endif
```

```
<C 声明放在此处>
```

```
#if defined(__cplusplus)
```

```

}
endif

```

3.5 预处理运算符

有时你只是想不用宏的值就知道一个宏是否被定义了。例如，如果你只支持两个编译器，那么在代码中可能会出现下面这样的代码段：

```

#if defined(__MSCVER)
<把微软标准语句放在此处>
#elif defined(__BCPLUSPLUS__)
<将 Borland 描述的语句放在此处>
#else
#error Compiler not supported.
#endif

```

如果 **defined** 运算符的目标在符号表中出现的值为 1，这意味着它是以前 **#define** 指令的主体或是编译器提供它作为系统预定义宏。**#error** 指令向 **stderr** 打印它的信息并停止翻译进程。没有必要为宏赋值，例如，要想在程序里加入调试跟踪输出，可以像下面这样做：

```

#if defined(DEBUG)
fprintf(stderr, "x = %d\n", x);
#endif

```

要定义 **DEBUG** 宏，只须在第一次使用宏之前加入下面的语句：

```
#define DEBUG
```

有如下的等价语句：

```

#if defined(X)    < == >    #ifdef X
#if !defined(X)   < == >    #ifndef X

```

defined 运算符比其右边的等价指令要灵活的多，因为可以把许多测试联合起来构成一个表达式：

```
#if defined(__cplusplus) && !defined(DEBUG)
```

defined 运算符是三个预处理运算符中的一个（参见表 3.3）。**#** 运算符，即字符串化（Stringizing）运算符，在引号中包括一个宏参数。如程序清单 3.3 中程序所描述的，这对调试是很有用的。**trace** 宏在括号中包括了宏参数，因此它们成为 **printf** 格式语句的一部分。例如，表达式 **trace(i,d)** 变为

```
printf("i" " = % d" "\n", i);
```

表 3.3

预处理运算符

运 算 符	用 途
#	字符串化
##	加标记
Defined	符号表查询

程序清单 3.3 说明字符串化运算符

```

/* trace.c: 说明用于调试的跟踪宏 */

#include <stdio.h>

#define trace(x,format) \
    printf("#x " = %" #format "\n",x)

main()
{
    int i = 1;
    float x = 2.0;
    char *s = "three";

    trace(i,d);
    trace(x,f);
    trace(s,s);
}

/* 输出:
i = 1
x = 2.000000
s = three
*/

```

并且，在编译器联结相邻的字符串之后，得到这种形式：

```
printf("i=%d\n",i);
```

如果不用字符串化运算符就无法创建这样的引号字符串，这是因为预处理器忽略了引号字符串中的宏。加标记（token-pasting）运算符`##`将两个标记联结在一起形成单个的标记，在程序清单 3.4 中调用 `trace2(1)` 被翻译成：

```
trace(x1,d)
```

这两个运算符周围任何的空格均被忽略。

程序清单 3.4 举例说明加标记运算符

```

/* trace2.cpp: 举例说明一个为调试用的 trace 宏 */
#include <stdio.h>

#define trace(x,format) \
    printf("#x " = %" #format "\n",x)
#define trace2(i) trace(x ## i,d)

main()

```

```
{
    int x1 = 1, x2 = 2, x3 = 3;
    trace2(1);
    trace2(2);
    trace2(3);
}
```

/* 输出:

x1 = 1

x2 = 2

x3 = 3

*/

3.6 实现 assert

实现 assert 揭示了关于宏的一个重要事实。由于 assert 的动作取决于测试的结果，因此可以先试一试“if”语句：

```
#define assert(cond) \
    if(!(cond)) _assert(#cond, _FILE_, _LINE_)
```

其中函数 _assert 打印消息并中断程序。因此，当 assert 在 if 语句之中发现自己时，这就产生了一个问题：

```
if (x>0)
    assert(x!=y);
else
    /* 无论是什么*/
因为它扩展为：
if (x>0)
    if (!(x!=y)) _assert("x!=y", "file.c", 7);
else
    /*无论是什么*/
```

这个缩排会使人产生误解，因为第二个 if 中断了其余的部分：

```
if(x>0)
    if (!(x!=y)) _assert("x!=y!", "file.c", 7);
else    /*糟糕！新的控制流！*/
    /*无论是什么*/
```

通常，为处理这种嵌套 if 问题采用如下的括号方式：

```
# define assert(cond) \
    if(!(cond))_assert(#cond, _FILE_, _LINE_)
但它的展开式为：
if(x>0)
    {if (!(x!=y))_assert("x!=y", "file.c", 7);}
else
    /*无论是什么*/
```

它和第二行的组合符} 生成了一个空语句使外部的 if 完整，而留下一个单独的 else，这是语法错误。正确定义 assert 的方式在程序清单 3.5 和程序清单 3.6 中给出。总之，当宏必须作出选择时，好的做法是把它写为一个表达式而不是语句。

程序清单 3.5 assert 宏的实现（无 NDEBUG）

```
/* assert.h */
#ifndef ASSERT_H
#define ASSERT_H

extern void __assert(char *, char *, long);

#define assert(cond) \
    ((cond) \
     ? (void) 0 \
     : __assert(#cond, __FILE__, __LINE__))

#endif
```

程序清单 3.6 __asser 支持函数

```
/* xassert.c */
#include <stdio.h>
#include <stdlib.h>

void __assert(char *cond, char *fname, long lineno)
{
    fprintf(stderr,
            "Assertion failed: %s, file %s, line %ld\n",
            cond, fname, lineno);
    abort();
}
```

3.7 宏的魅力

如果不能准确理解预处理器按照什么步骤来扩展宏，就会对一些不可思议的现象感到惊奇。例如，如果你在程序清单 3.4 的开始处插入下面一行：

```
#define x1 SURPRISE!
```

那么 `trace2(1)` 被扩展为：

```
trace(x ## 1,d)
```

它接着变为：


```
trace (x1,d)
```

但是预处理并不停止。它重新扫描该行看是否有任何其他的宏需要扩展，编译器所看到的程序文本的最后状态如程序清单 3.7 所示。

为进一步说明，考虑程序清单 3.8 中的文本。顺便说一下，它不是一个完整的程序，仅仅是为了预处理而已——不要试图用任何方法对它进行编译（如果是 Borland C++ 平台，可以使用 CPP 命令）。预处理器的输出见程序清单 3.9，str 宏仅仅是将它的参数加上引号，xstr 似乎是冗余的。

程序清单 3.7 令人吃惊的预处理源

```
main()
{
    int SURPRISE1 = 1, x2 = 2, x3 = 3;
    printf("x1 " = %" "d" "\n",SURPRISE!);
    printf("x2 " = %" "d" "\n",x2);
    printf("x3 " = %" "d" "\n",x3);
    return 0;
}
```

程序清单 3.8 说明宏的重新扫描

```
/* preproc.c: 测试#和##预处理运算符
 *
 * 注意：不要编译！只是预处理！
 */

/* 方便的字符串化宏 */
#define str(s) #s
#define xstr(s) str(s)

/* 方便的加标志宏 */
#define glue(a,b) a##b
#define xglue(a,b) glue(a,b)

/* 一些定义 */
#define ID(x) "This is version " ## xstr(x)
#define INCFILE(x) xstr(glue(version,x)) ".h"
#define VERSION 2
#define ION ATILE

/* 扩展某些宏 */
str(VERSION)
xstr(VERSION)
glue(VERSION,3)
xglue(VERSION,3)
glue(VERS,ION)
```

```
xglue(VERS, ION)

/* 扩展更多的宏*/
ID(VERSION)
INCFE(VERSION)
str(INCFE(VERSION))
xstr(INCFE(VERSION))
```

程序清单 3.9 程序清单 3.8 的预处理结果

```
"VERSION"
"2"
VERSION3
23
2
VERSATILE

"This is version "2"
"version2" ".h"
"INCFE(VERSION)"
"\version2\" ".h\""
```

但是它和 str 之间有一点很重要的不同。当然，语句 str(VERSION) 的输出是：

```
"VERSION"
```

而 xstr(VERSION) 扩展为：

```
str(2)
```

没有与#或##相连的参数在替代它们各自的参数之前就被完全展开了。然后语句被重新扫描，赋值为“2”。因此实际上 xstr 是 str 在引用自身参数前扩展其参数的一个版本。

glue 和 xglue 间也存在着同样的关系。语句 glue(VERSION,3) 把它的参数连接到符号 VERSION3，但 xglue(VERSION,3) 先扩展 VERSION 为：

```
glue(2, 3)
```

依次重新扫描成符号 23。下面两个语句有点微妙：

```
glue(VERS, ION)
== VERS ## ION
== VERSION
== 2
```

和

```
xglue(VERS, ION)
== glue(VERS, ATILE)
== VERS##ATILE
== VERSATILE
```

当然，如果 VERSATILE 是一个已定义的宏，则它可以进一步扩展。程序清单 3.8 的最后 4 个语句扩展如下：

```
ID (VERSION)
== "This is version " xstr(2)
== "This is version" str(2)
== "This is version" "2"
```

```
INCFILE(VERSION)
== xstr(glue(version,2)) ".h"
== xstr(version2) ".h"
== "version2" ".h"
```

```
str(INCFILE(VERSION))
== #INCFILE(VERSION)
== "INCFILE(VERSION)"
```

```
xstr(INCFILE(VERSION))
== str("version2" ".h")
== #"version2" ".h"
== "\version2\" \" ".h\" "
```

出于很明显的理由，#运算符在所有的嵌入引号和反斜杠之前插入了退出字符。

预处理器的宏替换功能显而易见为你提供了很大的灵活性。但要记住以下两点限制：

1. 无论何时预处理器在其替代文本中遇到当前的宏，不管在进程中有多么深的嵌套，它都不做扩展，只是保持不变（否则进程将永远不终止！）。例如，给出如下定义：

```
#define F(f) f(args)
#define args a,b
```

F(g) 扩展为 g(a,b)，那么 F(F) 扩展成什么呢？（答案是 F(a,b)）。

2. 如果被充分扩展的语句相似于一条预处理指令（例如扩展以后结果是一条 #include 指令），它没被调用，而是逐字地留在程序文本中

3.8 字符集、三字符运算符和双字符运算符

用来写程序的字符集不必与程序中运行所使用的相同。对于非英语的应用也是这样。一个 C 翻译程序仅仅理解英语的文字、用作运算符和标点符号的图形字符（共有 29 个）以及一些控制字符（回车换行符、横向制表符、纵向制表符和换页）。这就是基本源字符集。执行字符集是定义实现的，但是必须包含代表警示符（'\a'）、回退符（'\b'）、回车符（'\r'）和换行符（'\n'）。

一个 C++ 翻译器也必须接受通用字符名，它以下列形式之一出现在源文本中：

```
\uNNNN
\UNNNNNNNN
```

这里的 N 代表一个十六进制的数字。第一种形式对应的 ISO 10646 编码是 0000NNNN（即由零引导的统一编码），而第二种形式对应的是 NNNNNNNN。

许多非美国键盘不支持基本源字符集的一些元素，这给编写 C 程序带来困难。为了克服这个障碍，标准的 C 定义了很多三字符运算符，它是不变字符集（ISO 646-1983）中三字

符一体的字符，在西方世界几乎随处可见。每个三字符运算符对应源字符集中的不在 ISO 646 中字符。例如，无论何时预处理器在你的源文件中的任何地方（甚至在字符串里面）遇见标志 `??=`，都会用 `#` 字符的编码来替换。程序清单 3.11 的程序说明了怎样用三字符运算符写程序清单 3.10 的程序“Hello, world!”。

表 3.4 **三字符运算符序列**

三字符运算符	C 源字符
??	#
??([
??/	\
??)]
??'	^
??<	{
??!	
??>	}
??-	~

程序清单 3.10 “Hello, world!”程序

```

/* hello.c: 向用户或向世界问候 */
#include <stdio.h>

main(int argc, char *argv[])
{
    if (argc > 1 && argv[1] != NULL)
        printf("Hello, %s!\n", argv[1]);
    else
        printf("Hello, world!\n");
    return 0;
}

```

程序清单 3.11 使用三字符运算符的“Hello, world!”程序

```
/* hello2.c: 用三字符的问候程序 */
#include <stdio.h>

main(int argc, char *argv??(??))
??<
    if (argc > 1 && argv??(1??) != NULL)
```

```

    printf("Hello, %s!??/n",argv??(1??));
else
    printf("Hello, world!??/n");
return 0;
??>

```

为了努力开发更广泛可读性更强的程序，C++为非 ASCII 码开发者定义一套双字符运算符集和新的保留字集（见表 3.5）。程序清单 3.12 展示了使用这些新符号后的“Hello,world!”程序。也许你认为对称的括号运算符比三字符运算符更好看。

表 3.5 新的 C++双字符运算符和保留字

标 志	翻 译
<%	{
%>	}
<:	[
:>]
%%	#
Bitand	&
And	&&
Bitor	
Or	
Xor	^
Compl	~
and_eq	&=
or_eq	=
xor_eq	^=
Not	!
not_eq	!=

程序清单 3.12 带有新的 C++双字符和标记的“Hello,world”程序

```

// hello3.cpp: 使用 C++双字符的问候程序
#include <cstdio>
using namespace std;

```

```
main(int argc, char *argv<:1:> =
<%
    if (argc > 1 and argv<:1:> != NULL =
        printf("Hello, %s!??/n",argv<:1:> =;
    else
        printf("Hello, world!??/n");
    return 0;
%>
```

3.9 翻译阶段

标准 C 和 C++ 定义了 9 个不同的翻译阶段。当然，实现没有必要在代码中分成 9 个独立的阶段，但是翻译的结果必须好像已经这样做了一样，这 9 个阶段是：

1. 物理源字符被映射到源字符集中。其中包括三字符组合替换以及诸如把回车/换行映射到一个单独的 MSDOS 环境下的换行字符那样的东西。在 C++ 程序中，任何不在基础源字符集中的字符都被它的通用字符名替换。
 2. 所有以反斜杠结束的行都和它们接下来的行合并，并且删去反斜杠。
 3. 源码被分析成预处理标记，并且注释被一个单独的空字符所替换，C++ 双字符被识别为标记。
 4. 调用预处理指令并且扩展宏，对于任何被包含的文件循环地重复步骤 1 到 4。
 5. 源字符退出字符常量序列，普通字符名被映射成执行字符集成员（例如，‘a’将在 ASCII 环境下转换成 7 的一个字节值）。
 6. 相邻的字符串被连接。
 7. 传统的编译：词汇和语义分析，并翻译成汇编语言或机器码。
 8. （只有 C++）执行任何待解决的模板实例。
 9. 链接：解决外部引用，准备好程序映像以便执行。
- 预处理器由步骤 1 到步骤 4 组成。

3.10 小结

- 预处理器不能理解编程语言；
- 头不一定是文件；
- 彻底地记住带括号的宏；
- 宁可用内联函数而不用类函数的宏（除了字符串化和标记粘贴）；
- 宁可用常值而不要用类对象的宏；
- 用 assert 宏来捕捉不应该发生的概念错误；
- 有条件地用特殊的宏来编译头文件（来避免循环包含）；
- C 和 C++ 支持三字符组合以适应国际键盘，C++ 支持更多可读双字符和其他保留字。

C 标准库之一：面向合格的程序员

虽然看上去不太像，但 C 的确是一种非常小的语言。实际上，用现在的标准衡量，它首先是在非常小的平台上实现的，我的第一个 C 编译器就是在 Commodore 64 上运行的！C 的简单性和兼容性使它成为理想的系统编程语言和开发运行在嵌入式系统中的程序的理想编程语言，如用在车辆和照相机中。

C 在独立式环境和托管式环境中（如台式机或中等范围的计算机）一个非常重要的区别就在于是否有 C 标准库的存在。在独立式环境中，相容编译器仅需要在 `<float.h>`，`<limits.h>`，`<stdarg.h>` 和 `<stddef.h>` 中提供详细的类型和宏。从事典型数据处理工作的程序员认为库是理所当然存在的，实际上他们认为库是语言的一部分。日常的 C 代码大部分由库调用组成，甚至 I/O 程序如 `printf` 和 `scanf` 都是库的一部分，而不是语言。

标准 C 库由在 15 个头文件中声明的函数、类型定义和宏组成，每个头文件或多或少代表了一定范围的编程功能，例如 I/O 或字符串处理操作。为了方便起见，一些宏和类型定义如 `NULL` 和 `size_t` 不只在头文件中出现。无符号整型 `size_t` 可以保存 `sizeof` 运算符的结果，并可适用于数组索引。

我喜欢把标准库分成 3 组（见表 4.1~4.3）。第 I 组代表库的内容是每个（程序员都应该彻底了解的）。太多时候我看见很多程序“重新创造”基础库，如 `memcpy` 和 `strchr`，尽管如此，为了问心无愧地得到报酬，你也应该真正掌握第 II 组内容。虽然，你可能很少需要同时掌握第 III 组中的函数，但是你应该充分熟悉它们从而能在需要时知道怎样使用它们。

表 4.1 标准 C 头文件：第 I 组（每个 C 程序员都应该知道的知识）

<code><ctype.h></code>	字符处理
<code><stdio.h></code>	输入 / 输出
<code><stdlib.h></code>	复杂的工具
<code><string.h></code>	文本处理

表 4.2

标准 C 头文件：第 II 组（职业程序员的工具）

<code><assert.h></code>	支持有保护的程序的断言
<code><limits.h></code>	整型运算的系统参数
<code><stddef.h></code>	通用类型和常量
<code><time.h></code>	时间处理

表 4.3

标准 C 头文件：第 III 组（需要时可发挥很大作用的工具）

<code><errno.h></code>	错误检测
<code><float.h></code>	实数运算的系统参数
<code><locale.h></code>	文化自适应
<code><math.h></code>	数学函数
<code><setjmp.h></code>	非局部分支
<code><signal.h></code>	中断处理（从某种程度上说）
<code><stdarg.h></code>	可变长度参数表

我在本书中广泛地解释了标准 C 库，第 4 章到第 6 章是 C 库的总览，涉及一些容易忽视的函数。并提示出你可能没有意识到的行为。当需要更加详细的处理时，我会建议转到其他章节，而不是重复看这里的资料。说到标准 C 库详尽的参考，没有比 P.J.Plauger 的著作 *The Standard C Library*（Prentice-Hall, 1992）更好的了。

4.1 `<ctype.h>`

`<ctype.h>` 中的函数支持处理单个字符的典型操作（见表 4.4）。例如，为了确定一个字符 C 是否为大写，可用表达式 `isupper(c)`。许多老式的 C 程序布满了如下的表达式：

```
('A' <= c && c <= 'Z')
```

这样的文件可读性会很差。把这样的表达式放在宏帮助里，如：

```
#define ISUPPER(c) ('A' <= c & c <= 'Z')
```

表 4.4

`<ctype.h>` 中的函数

字符测试函数	
<code>isalnum</code>	字母与数字（是字母或者是数字）
<code>isalpha</code>	字母
<code>isctrl</code>	控制（当心）

续表

isdigit	“0”到“9”
isgraph	打印时可见
islower	小写字母
isprint	isgraph ‘ ’
ispunct	isgraph&&! isalnum
isspace	空格
isupper	大写字母
isxdigit	isdigit “a”到“f” “A”到“F”
字符映射函数	
tolower	转换成小写字母（如果符合要求）
toupper	转换成大写字母（如果符合要求）

但是这会使有副作用的表达式（如 ISUPPER（C++））不可靠，因为它的参数增加了两次。当然测试是否在“A”和“Z”之间时只给出了所期望的带有连续编码字符的字符集的结果，如 ASCII。<ctype.h>中的字符分类函数是安全的并且在所有的平台上都适用。

避免总是倾向于认为 ASCII 是执行字符集是很重要的。例如，虽然 ASCII 控制字符构成了编码 127 和那些小于 32 的编码，但是只有 7 个控制字符在所有的环境中都具有惟一的特征：警报（‘\a’）、退格（‘\b’）、回车（‘\r’）、进纸（‘\f’）、水平定位（‘\t’）、换行（‘\n’）、垂直定位（‘\v’）。改变定位时惟一不改变行为的两个函数是 isdigit 和 isxdigit（关于定位的更详细的内容请参阅第 6 章）。

虽然可以假设数字“0”到“9”在所有的 C 执行字符集中有连续的编码，但是具有字母字符的十六进制数则不然。程序清单 4.1 中的函数 atox 说明了如何把一个十六进制字符串转换成整型值。但是它只适用于类 ASCII 字符集。说明：

```
digit = toupper(*s) - 'A' + 10;
```

它不能保证表达式

```
toupper(*s) - 'A'
```

能给出正确的结果。程序清单 4.2 中的版本可以在任何平台上运行，因为它把所有的十六进制数相邻地存储在它自己的数组中，它用 strchr 搜索数组然后用指针运算来计算数字的值。

程序清单 4.1 在 ASCII 环境下将十六进制字符串转换成数字

```
#include <ctype.h>
```

```
long atox(char *s)
{
    long sum;

    /* 跳过空格 */
    while (isspace(*s))
        ++s;

    /* 做变换 */
    for (sum = 0L; isxdigit(*s); ++s)
    {
        int digit;
        if (isdigit(*s))
            digit = *s - '0';
        else
            digit = toupper(*s) - 'A' + 10;
        sum = sum*16L + digit;
    }
    return sum;
}
```

程序清单 4.2 程序清单 4.1 的简洁版本

```
#include <ctype.h>
#include <string.h>

long atox(char *s)
{
    char xdigs[] = "0123456789ABCDEF";
    long sum;

    /* 跳过空格 */
    while (isspace(*s))
        ++s;

    /* 做变换 */
    for (sum = 0L; isxdigit(*s); ++s)
    {
        int digit = strchr(xdigs, toupper(*s)) - xdigs;
        sum = sum*16L + digit;
    }

    return sum;
}
```

程序清单 4.3 用 sscanf 把十六进制字符串转换成数字

```
#include <stdio.h>

long atox(char *s)
{
    long n = 0L;
    sscanf(s, "%lx", &n);
    return n;
}
```

4.2 <stdio.h>

程序清单 4.1 和程序清单 4.2 的作者如果稍微更好地理解 scanf 的格式，则可以避免许多麻烦。如程序清单 4.3 所示，格式符 %x 做了所有读取十六进制数的工作。与前两个版本的程序不同，它甚至还处理了前置的加号或减号。scanf 和 printf 包含了许多被编程人员所忽视的特性（关于这两个函数特性的更多细节，请参阅第 17 章）。

表 4.5 所示的 scanf 和 printf 函数族执行格式化 I/O。更进一步地说，它们为 3 种类型的流提供了以下工具：标准流、文件流和字符串（即内存中的）流。不同类型的流格式化操作完全相同，但是所需的函数名和调用顺序存在不同程度的差异。

C 标准库中 <stdio.h> 组件还提供另外两类输入/输出工具：字符 I/O 和块 I/O（见表 4.6 和 4.7）。程序清单 4.4 和 4.5 中的函数分别用字符 I/O 和块 I/O 函数来完成一个文件到另一个文件中的拷贝。注意由于 fread 不能返回错误码，所以必须显式调用 ferror 来检测读取错误。

表 4.5 <stdio.h> 中定义的格式化 I/O 函数

固定长度参数列表	
scanf	标准输入
fscanf	文件输入
sscanf	内码输入
printf	标准输出
fprintf	文件输出
Sprintf	内码输出
可变长度参数列表	
vprintf	标准输出
vfprintf	文件输出
vsprintf	内码输出

表 4.6

<stdio.h> 中的字符 I/O 函数

单字符处理函数	
getchar	标准输入
getc	文件输入
ungetc	影响文件输入
fgetc	文件输入
putchar	标准输出
putc	文件输出
fputc	文件输出
字符串处理函数	
gets	标准输入
fgets	文件输入
puts	标准输出
fputs	文件输出

表 4.7

<stdio.h> 中其他的函数

块 I / O	文件定位
fread	fgetpos
fwrite	fsetpos
用文件名操作	fseek
remove	ftell
rename	rewind
临时文件	错误处理
tmpfile	clearerr
tmpnam	feof
文件访问函数	ferror
fopen	
freopen	
fclose	
fflush	
setbuf	
setvbuf	

如表 4.7 所示，<stdio.h>为文件定位提供函数。耗时函数 `fseek` 和 `ftell` 只能可靠地处理二进制模式打开的文件并且其位置要用一个长整型来限制。为了克服这些限制，ANSI 委员会发明了 `fgetpos` 和 `fsetpos`，它们用依赖平台抽象类型 `fpos_t` 作为文件位置指示标志（见表 4.8）。

程序清单 4.4 一个通过字节 I/O 拷贝文件的函数，错误时返回 EOF

```
/* copy1.c */
#include <stdio.h>

int copy(FILE *dest, FILE *source)
{
    int c;

    while ((c = getc(source)) != EOF)
        if (putc(c, dest) == EOF) /* 输出错误 */
            return EOF;
    return 0;
}
```

程序清单 4.5 一个通过块 I/O 拷贝文件的函数

```
/* copy2.c */
#include <stdio.h>

int copy(FILE *dest, FILE *source)
{
    size_t count;
    static char buf[BUFSIZ];

    while (!feof(source))
    {
        count = fread(buf, 1, BUFSIZ, source);
        if (ferror(source))
            return EOF;
        if (fwrite(buf, 1, count, dest) != count)
            return EOF;
    }
    return 0;
}
```

程序清单 4.6 中的程序很好地将文件定位函数应用于一种简单的四方向滚动大型文件浏览器。它仅在内存中保存一个屏幕大小的文本内容。如果想向上或下滚动文件，它会读取（或重读）并显示相邻的文本。若向下滚动（即向前）文件，屏幕上的数据文件位置被压入堆栈

中，程序从当前文件位置读下一个满屏。向上滚动时，它从栈中恢复以前屏幕的文件位置。第 19 章有这个程序的完整版本并非常详细地说明了文件 I/O。

表 4.8

<stdio.h>中定义的类型和宏

类型	
FILE	封装文件访问信息
fpos_t	文件位置（由 fgetpos 返回）
宏	
NULL	空指针
EOF	代表文件结束的特殊值
BUFSIZ	首选流缓冲区大小
FOPEN_MAX	同时打开文件的最大数
FILENAME_MAX	文件名的字符数最大值减 1
L_tmpnam	临时文件名字符数最大值减 1
TMP_MAX	从 tmpnam 返回的不同文件名的最多个数
SEEK_CUR	fseek 相对当前位置寻找的信号
SEEK_END	fseek 从文件结尾位置寻找的信号
SEEK_SET	fseek 从文件开始位置寻找的信号

程序清单 4.6 说明文件定位的文件浏览程序的概貌

```

/* view.c: 简单的 4 方向滚动文件浏览器 */

/* 省略 cls(), display(), read_a_screen() */

main(int argc, char *argv[])
{
    fpos_t top_pos, stk_[MAXSTACK];
    /* 省略详细资料 */
    top:
        /* 显示最初的屏幕 */
        rewind(f);
        fgetpos(f, &top_pos);
        /* 省略详细资料 */
        for (;;)
        {
            switch(c = toupper(getchar()))

```

```

{
    case 'D': /* 显示下一屏 */
        if (!feof(f))
        {
            PUSH(top_pos);
            fgetpos(f,&top_pos);
            read_a_screen(f);
            display(file);
        }
        break;continued
    case 'U': /* 显示前一屏 */
        if (stkptr_ > 0)
        {
            top_pos = POP();
            fsetpos(f,&top_pos);
            read_a_screen(f);
            display(file);
        }
        break;

    case 'T': /* 显示第一个屏幕 */
        stkptr_ = 0;
        goto top;

    case 'B': /* 显示最后的屏幕 */
        while (!feof(f))
        {
            PUSH(top_pos);
            fgetpos(f,&top_pos);
            read_a_screen(f);
        }
        display(file);
        break;

    case 'Q': /* 退出 */
        cls();
        return EXIT_SUCCESS;
}
/* 省略详细资料*/
}
}

```

4.3 <stdlib.h>

头文件<stdlib.h>具有一定的总结性。它定义了类型、宏和各种函数，这些函数用于内存

管理、排序和查找、整型运算、字符串到数字的转换、伪随机数序列、与环境的接口以及把多字节字符串和字符转换成宽字符表示或反之（见表 4.9）。程序清单 4.7 中的程序使用了全部四种内存管理函数对文本文件进行排序。每当它指向字符的指针数组满了时，它就用 `realloc` 来扩展数组以保存原始的内容（有关内存管理的深入处理请参见第 20 章）。

表 4.9

<stdlib.h>声明

类 型	
<code>div_t</code>	<code>div</code> 返回的结构
<code>ldiv_t</code>	<code>ldiv</code> 返回的结构
常量	
<code>EXIT_FAILURE</code>	<code>exit</code> 的简洁错误码
<code>EXIT_SUCCESS</code>	<code>exit</code> 的简洁成功码
<code>RAND_MAX</code>	<code>rand</code> 的最大返回值
<code>MB_CUR_MAX</code>	多字节字符中的最大字节数
<code>NULL</code>	
字符串转换函数	
<code>atof</code> <code>strtod</code>	
<code>atoi</code> <code>strtol</code>	
<code>atol</code> <code>strtoul</code>	
随机数函数	
<code>rand</code>	返回下一个非随机数
<code>srand</code>	“生成”伪随机数的顺序
内存管理	
<code>calloc</code> <code>realloc</code>	
<code>malloc</code> <code>free</code>	
与环境的接口	
<code>abort</code> <code>getenv</code>	
<code>atexit</code> <code>system</code>	
<code>exit</code>	
查找和分类	
<code>bsearch</code>	
<code>qsort</code>	

整型数运算

abs labs

div ldiv

多字节字符函数

mblen mbctows

mbtowc wcstombs

wctomb

程序清单 4.7 按存储器可利用的最大空间对文件进行排序

```

/* sort.c */
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#define MAXLINES 512

int comp(const void *, const void *);

main()
{
    int i;
    size_t nlines, maxlines = MAXLINES;
    static char s[BUFSIZ];
    char **lines = calloc(maxlines, sizeof(char *));

    /* 读文件 */
    for (nlines = 0; fgets(s, BUFSIZ, stdin); ++nlines)
    {
        if (nlines == maxlines)
        {
            /* 数组增加另外的 MAXLINES */
            size_t tlines
            maxlines += MAXLINES;
            lines = realloc(lines, maxlines * sizeof(char *));
            assert(tlines);
            lines = tlines;
        }

        /* 存储此行 */
    }
}

```

```
        lines[nlines] = malloc(strlen(s)+1);
        assert(lines[nlines]);
        strcpy(lines[nlines],s);
    }

    /*排序*/
    qsort(lines,nlines,sizeof lines[0],comp);

    /* 打印, 释放内存 */
    for (i = 0; i < nlines; ++i)
    {
        fputs(lines[i],stdout);
        fflush(stdout);
        assert(!ferror(stdout));
        free(lines[i]);
    }
    free(lines);
    return 0;
}

/* qsort() 的比较函数: */
int comp(const void *p1, const void *p2)
{
    return strcmp(* (const char **) p1, * (const char **) p2);
}
```

qsort 函数可以排序任何类型甚至用户定义结构的数组。要做到这一点需要为 qsort 函数提供 4 个参数:

1. 在何处查找数组;
2. 有几个元素;
3. 每一个元素有多大;
4. 如何比较它们。

第 4 个参数是一个具有两个指针的函数, 当第一个指针所指向的内容在第二个指针之前时, 该函数返回负数, 反之将返回正数, 其他情况时返回 0。下面的程序对整型数组进行排序:

```
/*qsort.c: 举例说明 qsort*/

#include <stdlib.h>
#include <stdio.h>

int comp(const void*,const void*);

main( )
{
```

```

int i;
int a[ ]={34,75,78,123,98};
int n=sizeof a/ sizeof a[10];
qsort(a,n, sizeof a[0],comp);
for (i=0;i<=n;++i)
    printf("%d", a[i]);
putchar('\n');
return 0;
}

```

/*输出

34 75 78 98 123

*/

比较函数 **comp** 将指针显示转换到正确的类型，并返回它们的差：

```

int comp(const void*p1,const void*p2)
{
    const int* pi1 = ( const int*) p1;
    const int* pi2 = ( const int*) p2;
    return *pi1- *pi2;
}

```

如果想排序 C 类型的字符串数组，可以将 **comp** 函数定义为下面的形式：

```

int comp(const void* p1,const void* p2)
{
    const char** ps1=( const char**) p1;
    const char** ps2=( const char**) p2;
    return strcmp(*ps1, *ps2);
}

```

查找函数 **bsearch** 在已排序的表中查找关键字，当与 **qsort** 一起使用时要为其提供一个函数，此函数返回指向包含关键字数组的指针（见程序清单 4.8）。

程序清单 4.9 中的程序说明了 **<stdlib.h>** 中一些不太常用到的函数。它通过产生 0 到 51 之间的随机序列来“洗”一副 52 张的牌。**srand** 函数用当前时间和日期的编码来“生成”伪随机生成器。要决定数字所对应的花色和大小，将随机数除以 13，每组花色中有 13 张牌，除得的余数是牌的大小（0 到 12，对应 A 到 K），而商则表示花色。如下：

```

0=clubs      (草花)
1=diamonds   (方块)
2=hearts     (红心)
3=spades     (黑桃)

```

程序清单 4.8 用 **bsearch** 函数查找一个已排序的记录数组

```

/* search.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```
struct person
{
    char last[16];
    char first[11];
    char phone[13];
    int age;
};

static int comp(const void *, const void *);

main()
{
    int i;
    struct person *p;
    static struct person key = {"", "", "555-1965", 0};
    static struct person people[] =
        {{"Ford", "Henry", "555-1903", 98},
         {"Lincoln", "Abraham", "555-1865", 161},
         {"Ford", "Edsel", "555-1965", 53},
         {"Trump", "Donald", "555-1988", 49}};

    /* 排序 */
    qsort(people, 4, sizeof people[0], comp);

    /* 查找 */
    p = (struct person *) bsearch(&key, people, 4, sizeof people[0], comp);
    if (p != NULL)
    {
        printf(
            "%s, %s, %s, %d\n",
            p->last,
            p->first,
            p->phone,
            p->age
        );
    }
    else
        puts("Not found");
    return 0;
}

/* 比较函数：关键值是电话号码字段 */
static int comp(const void *x, const void *y)
{
    struct person *p1 = (struct person *) x;
```

```

    struct person *p2 = (struct person *) y;

    return strcmp(p1->phone, p2->phone);
}

/* 输出: */
Ford, Edsel, 555-1965, 53

```

程序清单 4.9 说明<stdlib.h>中分离随机数和整数除法的洗牌程序

```

/* deal.c: 用一副洗完的纸牌发一手牌 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define DECKSIZE 52
#define SUITSIZE 13

main(int argc, char *argv[])
{
    int ncards = DECKSIZE; /* 默认时, 发整副纸牌 */
    char deck[DECKSIZE]; /* 小整数的数组 */
    size_t deckp;
    unsigned int seed;

    /* 获得可选的牌手数 */
    if (argc > 1)
        if ((ncards = abs(atoi(argv[1])) % DECKSIZE) == 0)
            ncards = DECKSIZE;

    /* 用当前时间编码产生随机数发生器 */
    seed = (unsigned int) time(NULL);
    srand(seed);

    /* 洗牌 */
    deckp = 0;
    while (deckp < ncards)
    {
        int num = rand() % DECKSIZE;
        if (memchr(deck, num, deckp) == NULL)
            deck[deckp++] = (char) num;
    }

    /* 发牌 */
}

```

```

    for (deckp = 0; deckp < ncards; ++deckp)
    {
        div_t card = div(deck[deckp], SUITSIZE);
        printf(
            "%c(%c)%c",
            "A23456789TJQK"[card.rem],
            "CDHS"[card.quot],
            (deckp+1) % SUITSIZE ? ' ' : '\n'
        );
    }

    return 0;
}

/* 输出: */
A(C) 6(S) 7(C) 9(C) 3(H) 6(C) 8(D) 3(C) 6(D) 5(D) 2(H) A(S) 4(H)
8(C) 8(H) 6(H) J(S) 7(S) Q(C) 2(C) Q(H) K(H) 4(C) 5(S) T(H) Q(S)
9(H) T(D) T(S) 9(D) K(C) 3(S) J(C) 5(C) T(C) K(S) 7(D) 2(D) 4(S)
8(S) 5(H) A(D) 7(H) 3(D) Q(D) A(H) 2(S) J(D) 9(S) K(D) J(H) 4(D)

```

div 函数将商和余数的计算全部一次完成，并将结果保存在具有整型成员商 quot 和余数 rem 的结构体类型 div_t 中。

在某些实现中，scanf 族中的函数调用 strtol 函数将字符串转换为整数。然而，直接使用 strtol 就可以读取任何基为 2 至 35 的数，如程序清单 4.10 中的程序所示。strtol 通过第二个参数更新 nextp，这样就可以通过字符串一个数一个数地转换。strtol 函数和 strtod 函数对 unsigned_long 和 double 型数据具有类似的特点。用 strtol 函数可以编写更高级的 atox 转换函数，如程序清单 4.11 所示。

程序清单 4.10 用 strtol 函数读不同基的数

```

#include <stdio.h>
#include <stdlib.h>

main()
{
    char *input = "101 123 45678 90abc g";
    char *nextp = input;
    long bin, oct, dec, hex, beyond;

    bin = strtol(nextp, &nextp, 2);
    oct = strtol(nextp, &nextp, 8);
    dec = strtol(nextp, &nextp, 10);
    hex = strtol(nextp, &nextp, 16);
    beyond = strtol(nextp, &nextp, 17);
}

```

```

    printf("bin = %ld\n",bin);
    printf("oct = %lo\n",oct);
    printf("dec = %ld\n",dec);
    printf("hex = %lx\n",hex);
    printf("beyond = %ld\n",beyond);
    return 0;
}

```

```

/* 输出: */
bin = 5
oct = 123
dec = 45678
hex = 90abc
beyond = 16

```

程序清单 4.11 使用 strtol 函数的更好的 atox 版本

```

#include <stdlib.h>

long atox(char *s)
{
    return strtol(s,NULL,16);
}

```

常用的 `exit` 和 `abort` 函数提供了两种终止程序的方法。`abort` 函数突然地终止程序执行而不承担任何清除工作，例如关闭打开的文件。而无论何时调用 `exit` 函数都终止程序的执行，但所有打开的文件都会被关闭，所有临时文件都被删除，并且向 `exit` 函数传递的整型参数被返回到操作系统，此外，如果需要在 `main` 函数之后执行也可以调用 `exit` 函数。然而，`exit` 做的第一件事就是调用任何一个已经被用户用 `atexit` 注册的 `exit` 处理器，方法如下：

```

void my_handler(); /*exit 处理器函数*/
atexit (my_handler);

```

`exit` 处理器函数不能带有参数，并且要返回 `void` 类型值。对于一般的 `exit` 处理（即从 `main` 函数返回或调用 `exit` 函数），所有的处理器函数按照注册时的逆序被调用。最多可以注册 32 个 `exit` 处理器函数。

`getenv` 函数允许在主机环境查询字符串。例如，要找到路径变量 `PATH` 的当前设置，这在很多环境下是很常见的，可以这样做：

```

char *path =getenv("PATH");

```

指针指向程序外部的存储器，所以如果想保留当前的值，就必须在再次调用 `getenv` 函数之前把它拷贝到程序变量中。

4.4 <string.h>

<string.h>中定义的函数如表 4.10 所示。所有带有 str 前缀的函数需要空终止字符串参数，而带有 mem 前缀的函数处理未处理过的存储器。在程序清单 4.2 中已经见过 strchr 函数，在程序清单 4.9 中见过对应的 memchr。为了将未经处理的字节从一个位置移到另一个位置，可以用 memcpy，或者当源单元与目的单元缓冲器相交迭时用 memmove。从次数上判断，我已经看到 memcpy 函数在其他代码中被“彻底改造”，我相信那是标准库中最容易被疏漏的函数。

表 4.10

<string.h>中定义的函数

复制	
memcpy	strcpy
memmove	strncpy
连结	
strcat	
strncat	
比较	
memcmp	strcmp
strcmp	strxfrm
strcoll	
查找函数	
memchr	strchr
strchr	strspn
strcspn	strstr
strpbrk	strtok
其他函数	
memset	
strerror	
strlen	

字符串查找函数也经常闲置而没被使用。程序清单 4.12 中的程序用 strstr 函数从包含给定字符串的文档文件中抽取所有的行。<string.h>中的下面 3 个函数由于隐藏名字可能极少使用：

1. `size_t strspn(const char *s1, const char *s2)`

在 `s1` 中“延续”`s2` 中事件的字符。也就是说，返回 `s1` 中第一个在 `s2` 中不存在的字符的索引。标准 C++ 字符串类称这种操作为 `find_first_not_of`。

2. `size_t strcspn(const char *s1, const char *s2)`

在 `s1` 中“延续”非 `s2` 中事件的字符。换句话说，返回 `s1` 中第一个也在 `s2` 中存在的字符的索引。标准 C++ 字符串类称这种操作为 `find_first_of`。

3. `char *trcspn(char *s1, const char *s2)`

类似 `strcspn` 函数，但返回的是指针而不是索引。

程序清单 4.13 中的程序说明了这些函数。更多有关字符串的例子请参阅第 17 章。

程序清单 4.12 采用 `strstr` 查找子字符串

```
/* find.c: 从文件抽取行 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

main(int argc, char *argv[])
{
    char line[BUFSIZ];          /* 关于 BUFSIZ 请参见表 4.7 */
    char *search_str;
    int lineno = 0;

    if (argc == 1)
        return EXIT_FAILURE; /* 查找所需字符串! */
    else
        search_str = argv[1];

    while (gets(line))
    {
        ++lineno;
        if (strstr(line, search_str))
            printf("%d: %s\n", lineno, line);
    }

    return EXIT_SUCCESS;
}

/*命令“find str <find.c”的结果: */
4: #include <string.h>
11:     char *search_str;
15:         return 1; /* 查找所需字符串 */
17:         search_str = argv[1];
22:         if (strstr(line, search_str))
```

程序清单 4.13 说明所选的字符串查找函数

```
#include <stdio.h>
#include <string.h>

void display_span(char *, int);

main()
{
    char *s = "Eeek! A mouse device!";
    char *vowels = "AEIOUaeiou";
    char *punct = "`~!@#$%^&*()-_+=\\|[]{};: '\",.<.>/?";
    char *ptr;

    display_span(s, strspn(s, vowels));
    display_span(s, strspn(s, punct));
    display_span(s, strcspn(s, vowels));
    display_span(s, strcspn(s, punct));

    ptr = strpbrk(s, vowels);
    puts(ptr);

    ptr = strpbrk(s, punct);
    puts(ptr);

    return 0;
}

void display_span(char *s, size_t index)
{
    printf("%d characters spanned: %.*s\n",
        index, index, s);
}

/* 输出: */
3 characters spanned: Eee
0 characters spanned:
0 characters spanned:
4 characters spanned: Eeek
Eeek! A mouse device!
! A mouse device!
```

C 标准库之二：面向熟练的程序员

在上一章中，我把 C 标准库的 15 个头文件分成了 3 组，每组都代表了不同的掌握程度（见第 4 章表 4.1～表 4.3）。本章探讨第二组，如果使用的话，会在一定程度上使编程增色。

5.1 <assert.h>

在组织得很好的程序中，有许多可以进行断言的关键点，例如指向下一个开放数组元素的索引。在开发阶段测试这些断言，并且为程序维护员（当然，程序开发人员也常常需要进行维护的工作）将其存档是很重要的。C 标准库出于这个目的提供了断言（assert）宏。例如，可以将以上的断言表示如下：

```
#include <assert.h>
...
assert(nitems<MAXITEMS && i== nitems);
```

如果条件满足，就可以继续执行程序。否则 assert 会打印一条消息说明条件、文件名以及行号，然后调用 abort 函数来终止程序。

应该使用 assert 来验证程序的内部逻辑。如果假设执行的某一线程是不可能的，那么就调用 assert(0) 来表示这个问题，如下所示：

```
switch(color)
{
case RED:
...
case BLUE
...
case GREEN:
...
}
```

```
default:
    assert(0);
}
```

宏 `assert` 用于检验参数也是很方便的。例如，要对一个以字符串为参数的函数进行参数检验，可以这样做：

```
char *f(char *s)
[
    assert(s);
    ...
}
```

当然，断言是用来抓住逻辑错误的，而不是运行期错误。例如：由用户或环境创建的非运行期条件应该曾经生成过空指针；那很显然是你的问题，所以可能想在这种情况下中使用 `assert`（当然，也可在调试的时候找出它们）。像内存故障这样的运行期状况要求更强大、更强壮的异常处理机制，不适合用断言（`assert`）来处理。

当代码准备生成时，应该关闭断言处理（因为已经找出了所有的错误）。要这样做，可以在每个翻译单元开始的附近包括语句：

```
#define NDEBUG
```

或者如果编译器允许的话（多数使用 `-D` 开关），可以在命令行里定义宏。使用 `NDEBUG` 定义，所有的断言都扩展成了空宏，但是文本保留在文件用的代码中。请确定没有在 `assert` 宏中放入必要的程序操作。

5.2 <limits.h>

理想情况下，可移植的程序并不直接依赖于任何一个环境中的细节。甚至可以设想所有 8 位组成的字节都不安全。头文件 `<limits.h>` 为所有的整数类型定义了上下限（见表 5.1）。程序清单 5.1 中的程序触发一个整数中每一位的开关。这个程序使用在 `<limits.h>` 中定义的 `CHAR_BIT` 值作为字节中的位数，来决定在整数中的位数。如程序清单 5.2 所示，也可以使用 `<limits.h>` 来决定必须跨越一定范围有符号数值所用最有效的数据类型。

表 5.1 触发整型数的每一位

<code>CHAR_BIT</code>	8
<code>SCHAR_MIN</code>	-127
<code>SCHAR_MAX</code>	127
<code>UCHAR_MAX</code>	255
<code>[If char == signed char]</code>	

续表

CHAR_MIN	SCHAR_MIN
CHAR_MAX	SCHAR_MAX
[else]	
CHAR_MIN	0
CHAR_MAX	UNCHAR_MAX
[end if]	
MB_LEN_MAX	1
SHRT_MIN	-32767
SHRT_MAX	32767
USHRT_MAX	65535
INT_MIN	-32767
INT_MAX	32767
UINT_MAX	65535
LONG_MIN	-2147483647
LONG_MAX	2147483674
ULONG_MAX	4294967295

程序清单 5.1 使用在<limits.h>中定义的 CHAR_BIT 值

```

/* bit3.c: 在一个词中触发位*/
#include <stdio.h>
#include <limits.h>

#define WORD      unsigned int
#define NBYTES    sizeof(WORD)
#define NBITS     (NBYTES * CHAR_BIT)
#define NXDIGITS  (NBYTES * 2)

```

```
main()
{
    WORD n = 0;
    int i, j;

    for (j = 0; j < 2; ++j)

        for (i = 0; i < NBITS; ++i)
        {
            n ^= (1 << i);
            printf("%0*X\n", NXDIGITS, n);
        }

    return 0;
}
```

/* 输出: */

0001
0003
0007
000F
001F
003F
007F
00FF
01FF
03FF
07FF
0FFF
1FFF
3FFF
7FFF
FFFF
FFFE
FFFC
FFF8
FFF0
FFE0
FFC0
FF80
FF00
FE00
FC00
F800
F000

```
E000
C000
8000
0000
```

程序清单 5.2 使用<limits.h>选择一个合适的数字类型

```
/* range.c */
#include <stdio.h>
#include <limits.h>

#define LOWER_BOUND <your min here>
#define UPPER_BOUND <your max here>

/* 为范围确定最小的数字的类型*/
#if LOWER_BOUND < LONG_MIN || LONG_MAX < UPPER_BOUND
    typedef double Num_t;
#elif LOWER_BOUND < INT_MIN || INT_MAX < UPPER_BOUND
    typedef long Num_t;
#elif LOWER_BOUND < SCHAR_MIN || SCHAR_MAX < UPPER_BOUND
    typedef int Num_t;
#else
    typedef signed char Num_t;
#endif

main()
{
    Num_t x;

    printf("sizeof(Num_t) == %d\n", sizeof x);
    return 0;
}
```

5.3 <stddef.h>

头文件<stddef.h>定义了3种类型的同义字和两个宏（见表5.2）。当对两个指向同一数组元素的指针进行相减时（包括一个位置超过数组末尾的情况），获得两个相应下标的差，其大小是两个指针之间元素的个数。结果的类型不是整型就是长整型，无论哪个都适合你的环境。头文件<stddef.h>定义了适合该种操作的类型如 ptrdiff_t。

运算符 sizeof 返回类型 size_t 的值，它是一个无符号整型数，代表了在开发环境中所能声明的最大数据对象的大小（通常是无符号整型（unsigned int）或无符号长整型（unsigned long））。它总是用于与 ptrdiff_t 类型极为相似的无符号类型。如果浏览 C 标准库中的头文件，

就会发现这些文件广泛地使用 `size_t` 类型。对于所有的数组索引和指针算术来说，使用 `size_t` 是个好主意（即为指针增加偏移量），除非由于某种原因需要在 0 以下进行倒计，这是无符号的整数无法做到的。

表 5.2

<stddef.h> 中的定义

类型同义字	
<code>ptrdiff_t</code>	指针相减的类型
<code>size_t</code>	<code>sizeof</code> 的类型
<code>wchar_t</code>	宽字符类型
宏	
<code>NULL</code>	0 指针
<code>offsetof</code>	结构体成员的字节偏移量

`wchar_t` 类型包含一个宽字符，它是一个用来表示超出 ASCII 码标准之外的定义实现的整数类型。可以用预先定义好了的 `L` 来表示宽字符常量，如下：

```
#include <stddef.h>
wchar_t c = L 'a';
wchar_t *s= L "abcde";
```

如程序清单 5.3 所表明，我的环境定义了一个宽字符作为两字节整数，它可以和（国际标准的 16）位统一字符编码标准很好地兼容。列于表 5.3 中的<stdlib.h>函数使用 `wchar_t` 类型。标准化附录是 1993 年被标准 C 接受的官方附录，它为处理宽字符和多字节字符定义了许多附加的函数。想要了解更详细的信息，请参见 P.J. Plauger 发表在 *C/C++ User Journal* 的 1993 年 4 月和 5 月两期上的社论。

程序清单 5.3 说明宽字符串

```
/* wide.c */
#include <stddef.h>
#include <stdio.h>

main()
{
    char str[] = "hello";
    wchar_t wcs[] = L"hello";

    printf("sizeof str = %d\n",sizeof str);
    printf("sizeof wcs = %d\n",sizeof wcs);
    return 0;
}
```



```

}

/* 输出：*/
sizeof str = 6
sizeof wcs = 12

```

表 5.3 使用 wchar_t 的<stddef.h>函数

mbtowc	把多字节字符翻译成宽字节字符
wctomb	把宽字节字符翻译成多字节字符
mbstowcs	把多字节字符串翻译成宽字符串
wcstombs	把宽字符串翻译成多字节字符串

宏 NULL 是通用的 0 指针常量。应该包含定义 NULL 的头文件(即 `stddef.h`, `stdio.h`, `stdlib.h`, `string.h`, `locale.h`)以便让系统负责定义 NULL。当需要在编译单元而不是其他单元定义 NULL 时, 使用头文件 `<stddef.h>` 是很方便的。

宏 `offsetof` 返回从结构体开始到它某一成员的字节偏移量。由于地址队列的限制, 一些实现在结构体成员之间插入了未使用的字节, 所以不能认为成员的偏移量正好是在它前面的成员的大小之和。例如, 程序清单 5.4 表明 `Person` 结构在 `name` 成员之后有一个字节的间隔, 这允许成员 `age` 开始于一个字的分界(这里一个字是两个字节)。如果需要指向结构体成员的显式指针则可以使用 `offsetof`, 如下:

```

struct Person P;
int *age_p;
age_p = (int *) ((char *) &P + offset (struct Person , age));

```

程序清单 5.4 `offsetof` 说明一个结构中的排列

```

/* offset.c */
#include <stddef.h>
#include <stdio.h>
struct Person
{
    char name[15];
    int age;
};

main()
{
    printf("%d\n", offsetof(struct Person, age));
    return 0;
}

/* 输出：*/

```

5.4 <time.h>

大多数环境提供保持时间的某些机制。标准 C 提供 `clock_t` 类型，它是一个可以跟踪处理器时间的数字类型。`clock` 函数返回一个表示当前处理器时间的定义实现的 `clock_t` 类型的值。不幸的是，所谓“处理器时间”随工作平台而变化，所以 `clock` 本身并不是很有用。然而，可以比较处理器时间然后除以 `CLOCKS_PER_SEC` 常量，这样及时地给出两点之间消逝的秒数。程序清单 5.5、程序清单 5.6 和程序清单 5.7 的程序使用 `clock` 实现简单的秒表功能。

程序清单 5.5 秒表功能声明

```
/* timer.h */
void timer_reset(void);
void timer_wait(double nsecs);
double timer_elapsed(void);
```

程序清单 5.6 秒表功能实现

```
/* timer.c */
#include <time.h>
#include "timer.h"

static clock_t start = (clock_t) 0;

/* 复位计时器 */
void timer_reset(void)
{
    start = clock();
}

/* 等待若干秒 */
void timer_wait(double secs)
{
    clock_t stop = clock() +
                  (clock_t) (secs * CLOCKS_PER_SEC);
    while (clock() < stop);
}

/* 用秒计算逝去的时间 */
double timer_elapsed(void)
```

```

{
    return (double)(clock() - start) / CLOCKS_PER_SEC;
}

```

程序清单 5.7 说明秒表功能

```

/* t_timer.c:    测试秒表功能 */
#include <stdio.h>
#include <limits.h>
#include "timer.h"

main()
{
    long i;

    timer_reset();

    /* 延迟 */
    for (i = 0; i < LONG_MAX; ++i)
        ;

    /* 获得逝去时间 */
    printf("elapsed time: %lf secs\n", timer_elapsed());
    return 0;
}

/* 输出: */
elapsed time: 565.070000 secs

```

程序清单 5.8 <time.h>中 tm 的定义

```

struct tm
{
    int    tm_sec;    /* 秒 (0 - 60) */
    int    tm_min;    /* 分钟 (0 - 59) */
    int    tm_hour;    /* 小时 (0 - 23) */
    int    tm_mday;    /* 一个月的天数 (1 - 31) */
    int    tm_mon;    /* 月 (0 - 11) */
    int    tm_year;    /* 从 1900 开始的年 */
    int    tm_wday;    /* 一周的天数 (0 - 6) */
    int    tm_yday;    /* 一年的天数 (0 - 365) */
    int    tm_isdst;    /* 昼夜储蓄标志 */
};

```

<time.h>中其他的函数处理日历时间，time 函数返回依赖系统编码的当前时间及日期，作为 time_t 类型（通常是长整型）。localtime 函数把 time_t 解码成 tm 结构（见程序清单 5.8）。asctime 函数返回一个标准格式解码时间的文本表示，即：

```
Mon Nov 28 14:59:03 1994
```

表 5.4

<time.h>中的定义

宏	
NULL	
CLOCKS_PER_SEC	
类型	
size_t	
clock_t	系统时间
time_t	编码的时间日期值
struct tm	解码的时间日期成分
函数	
difftime	两个时间间隔
mktime	标准化结构 tm
time	重新得到当前时间的编码
asctime	时间值的文本表示
ctime	当前时间的文本表示
gmtime	解码成 UTC 时间
localtime	解码一个时间值
strftime	格式化一个解码时间

表 5.4 提供了<time.h>中的一系列定义。有关更多关于时间和日期处理的细节请参见第 19 章。

5.5 字符集

书写符号是以书面文本传递信息的一系列符号。世界上有 30 多种主要的书写符号。有

一些符号，如罗马和（古代）斯拉夫文用于许多语言中。按照表 5.5 中的层次结构可以给世界上的书写符号分类。

表 5.5

世界书写符号

欧洲文字

亚美尼亚文，（古代）斯拉夫文，格鲁吉亚文，希腊文，罗马文

印度文字**北部**

孟加拉文，梵文，古吉拉特文，果鲁穆奇文，奥里雅文

南部

埃纳德文，马来文，僧伽罗文，泰米尔文，泰卢固文

东南部

缅甸文，高棉文，老挝文，泰文

中亚

西藏（文）

中东地区文字

阿拉伯（文），希伯来文

东亚（东方的文字）

汉文，汉语拼音字母，假名，（平假名和片假名），朝鲜文

亚洲其他地区文字

兰那泰国文，曼吉安文，蒙古文，那克希文，玻拉德文，帕哈里文，蒙文，泰努文，泰奴阿文，彝文

非洲

埃塞俄比亚文，奥斯曼文，利比亚文，瓦伊文

美国土著

克里文

混合的文字符号

化学，数学，出版符号，IPA（国际语音字母表）

大多数书写符号是按字母排列的。然而，由中国人、日本人和韩国人使用的汉字符号是表意的（更准确地说是语标的）书写符号。每个汉字代表了一个对象或一个概念（我们称之为字，尽管多数汉字要求两个语标），在这些语言中没有用字母表中的字母组成字的概念。

字符集是带有相关数字编码的文本符号的集合。我们大多数人熟悉的 ASCII 字符集把我

们日常使用的字母和数字映射成在[32,126] 范围内的整数,并用特殊的控制代码来填充[0,127]范围的 7 个比特位。如首字母缩写词中的“A”表明的那样,这完全是美国标准。更进一步说,它只规定了在一个 8 位的单字节中可得到的 256 个代码点的一半。有许多扩展的 ASCII 字符集用图表字符、重音字母、或非罗马字符来添充上面的范围[128,255]。由于 256 个代码点不能覆盖目前使用的罗马字母,所以出现了 5 个分离的单字节应用标准用于使用罗马字母(见表 5.6)单字节字符集有一个明显的不足,就是难以在单一应用中同步地处理来自不同地区的数据(如来自希腊和希伯来),因为同样的代码点被不同的编码共享。单字节编码全然不适合中文、日文和韩文,因为有成百上千个汉字字符。

表 5.6

标准 8 位字符集

ISO 8859-1 (Latin-1)	Western European
ISO 8859-2 (Latin-2)	Eastern European
ISO 8859-3 (Latin-3)	Southeastern European
ISO 8859-4 (Latin-4)	Northern European
ISO 8859-5	Cyrillic
ISO 8859-6	Arabic
ISO 8859-7	Greek
ISO 8859-8	Hebrew
ISO 8859-9 (Latin-5)	Western European+Turkish

一种提高单编码字符数的方式是把字符映射成多个字节。多字节的字符集编码把一个字符映射为一个字节值或多个字节值的长度可变的序列。在常用的编码中,如果一个字节最显著的特征位是零,这个字符就是标准的 ASCII;如果不是,这个字节和下一个字节形成了本地字符的 16 位代码。多字节编码能有效存储,因为它们没有未使用的字节,但是它们也有不足,即要求用特殊的算法把索引计算成字符串或计算字符串的长度,因为字符是用变化的字节数来表示的。为了克服字符串索引的问题,标准 C 定义了处理多字节和可将多字节字符串转化为宽字符字符串的函数(即字符串 `wchar_t`,通常是两字节字符)。然而,这些多字节和宽字符函数一般只能在 XPG4-UNIX 平台和日文平台上实现。在 1993 年批准的 C 标准附录中定义了许多处理多字节和宽字符序列的附加函数,这应该吸引美国的开发商们跳出他们自我满足的文化圈。

5.6 代码页

因为标准 ASCII 码只由 128 个代码点组成,还有 128 多个代码点在 8 位字符编码中等待使用。普遍的作法是将上面部分的 128 个代码用于本地字符的使用。像这样对 128~255 之间

的值与 ASCII 一起进行编码，在 MS-DOS 和 Microsoft Windows 平台下被称为代码页。美国 and 大部分欧洲国家（#437）在 IBM PC 上的缺省代码页中包括一些流程图、其他图形字符和带有区别标志的罗马字符。其他 MS-DOS 代码页包括：

863	加拿大英文—法文
850	多语言（拉丁—1）
865	日耳曼文
860	葡萄牙文
852	斯拉夫文（拉丁—2）

因为代码页使用 [128, 255] 范围内的代码点，所以不依靠或修改程序数据中任何字节的高位值（这在传统的 UNIX 世界里是一个很普遍的做法）是很重要的。遵循这个规则的程序被称为 8 位清除。

5.7 字符集标准

世界上使用最广泛的字符集是 7 位 ASCII。从本质上讲 ISO 646 是具有一些留给本地化代码的 ASCII 码。例如，货币符号——代码点 0x24——仅在美国是 \$，为了符合当地惯例允许它进行“浮动”。ISO 646 有时被称为可移植的字符集（PCS），而且它是编程语言的标准字母表。

为了在单字节编码中利用所有可能的 256 个代码点，ISO 8859 定义了 9 个 8 位映射以容纳在表 5.6 中列出的字母语言。每个映射保留 ISO 646 作为子集，因此它们主要区别在于上面部分的 128 个代码点。一些代码点是 MS-DOS 代码页的基础。

在远东没有用于多字节字符集的官方 ISO 标准。然而它的每一个地区都有当地的国家标准。还有基于国家标准且广泛应用的工业标准，包括 E-Ten、Big Five 和 Shift JIS。

5.8 ISO 10646

为了简化国际化应用的发展，ISO 开发了通用 8 位组合编码字符集（ISO 10646）以在单编码中容纳所有有意义的现代语言的字符。octet 是一个邻近的、有序的 8 位集合，在大多数系统中是一个字节。ISO 10646 可允许 2 147 483 648 个字符，尽管其中只有 34 168 个被定义字符。它被分成 128 组，每组包含有 65536 个字符的 256 层面（256 行×256 列——见图 5.1）。

2^{31} 个字符中的任何一个可用 4 个 8 位组表示来寻址，分别是四维空间中的组、层面、行和列。因此，ISO 10646 是 32 位的字符编码，ASCII 码点是 ISO 10646 的一个子集：在前面加 0 来填满 32 位。例如，字母“a”的十六进制编码是 00000061（即 0 组、0 层面、0 行、0x61 列）。0 组的 0 层面是 32 768 级中惟——一个叫做基本多语言层面（Bmp）的。ISO 10646 允许一致实现是基于 BMP 的，就是说，只需要有两个 octets——分别在 BMP 中代表行和列。完全由四个 8 位形成的编码叫做 UCS-4，而两个 8 位形成的编码叫 UCS-2。因此在 UCS-2 下，

字母“a”的十六进制编码是 0061 (0 行, 0x61 列)。BMP 中第 0 行在本质上是用美元符号作为货币符号的 ISO 8859-1 (Latin-1)。

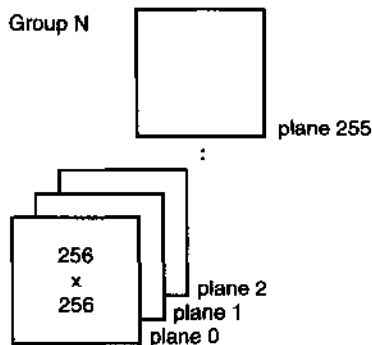


图 5.1 典型的 ISO 10646 组的拓扑图

ISO 10646 也定义了组合字符, 如不占空间的读音符号。在相容应用中, 组合的字符总是跟在它们修改的基本字符之后。那么, UCS-2 中“ä”的编码由两个 16 位整型数组成: 0061 与 0301 (0310 非空格敏锐)。为了同现存的字符集兼容, 还有“ä”的单一 UCS-2 码点 (00e1)。一般来说, 只有罗马字母才有这样的双重表示。一些非罗马语言, 如阿拉伯语、北印度语和泰语, 要求使用组合的字符。作为工具和应用有三个级别的相容性:

级别 1 不允许组合字符。

级别 2 只允许阿拉伯语、希伯来语及印度语的组合字符。

级别 3 对组合字符没有限制。

5.9 统一字符编码

统一字符编码 (Unicode) 是一个支持最现代书面语言的 16 位编码。它起源于 ISO 10646, 但是, 它从 Unicode 1.1 版本开始, 现在是 ISO 10646 的一个子集 (确切地说, 是 UCS-2, 级别 3)。开发商现在开始支持统一字符编码, 其工具也很快可以利用。支持 ISO 10646 32 位编码工具不需要再期待好多年, 尤其超越了 BMP 的无层面已经开始流行。统一字符编码标准也定义了映射图表以完成大多数本国或国际的字符集标准与和统一字符编码标准之间的互译。

有些应用应该很容易地转换成统一字符编码标准。既然 ASCII 是一个子集, 它只需要将窄字符 (8 位) 转化为宽字符。在 C 和 C++ 中, 这意味着用 `wchar_t` 声明代替 `char`。其他的字符集, 如泰国文和朝鲜文的字符集, 在统一字符编码中以相同的相关顺序出现, 所以只需加上或减去一个固定的偏移量。转换汉字字符则需要一个查找表。

C 标准库之三：面向优秀的程序员

这一章阐述的是在前两章中没有涉及的标准 C 头文件的功能。关于 C 标准库中所有头文件的分类，请参见第 4 章中的表 4.1~4.3。

6.1 <float.h>

在不同的计算机环境之间，可能没有其他内容能像浮点数系统那样变化得如此之大。一些平台要求有特殊的硬件来支持本平台的浮点指令，而其他的平台则通过软件来提供支持浮点运算的能力。

浮点数字系统是一个可以用一些固定数字的科学记数符号来表达的数字集合。准确地说，它是一系列有限数字的形式： $\pm 0.d_1d_2\dots d_p \times \beta^e$ ，其中 $m \leq e \leq M$ 。参数 β , p , m 和 M 分别代表了基数、精度、最小指数和最大指数。头文件<float.h>为这些浮点参数和其他重要的浮点参数提供了明确的常量（见表 6.1，以及附录 6.1 “浮点数系统”）。如表 6.1 所示，每一种实现有三个内在的单独的数字系统，它们是单精度、双精度和长双精度。

表 6.1 <float.h>中的定义

参 数	意 义	“最小”值
FLT_RADIX	指数的底	2
FLT_ROUNDS	取整模式	(*)
FLT_MANT_DIG	单精度数的精度	
DBL_MANT_DIG	双精度数的精度	

续表

参 数	意 义	“最小”值
LDBL_MANT_DIG	长双精度数的精度	
FLT_DIG	单精度数以 10 为底的精度	6
DBL_DIG	双精度数以 10 为底的精度	10
LDBL_DIG	长双精度数以 10 为底的精度	10
FLT_MIN_EXP	单精度数的最小指数	
DBL_MIN_EXP	双精度数的最小指数	
LDBL_MIN_EXP	长双精度数的最小指数	
FLT_MIN_10_EXP	单精度数以 10 为底的最小指数	-37
DBL_MIN_10_EXP	双精度数以 10 为底的最小指数	-37
LDBL_MIN_10_EXP	长双精度数以 10 为底的最小指数	-37
FLT_MIN	最小的单精度数	1E-37
DBL_MIN	最小的双精度数	1E-37
LDBL_MIN	最小的长双精度数	1E-37
FLT_MAX_EXP	单精度数的最大指数	
DBL_MAX_EXP	双精度数的最大指数	
LDBL_MAX_EXP	长双精度数的最大指数	
FLT_MAX_10_EXP	单精度数以 10 为底的最大指数	+37
DBL_MAX_10_EXP	双精度数以 10 为底的最大指数	+37
LDBL_MAX_10_EXP	长双精度数以 10 为底的最大指数	+37
FLT_MAX	最大的单精度数	1E+37

续表

参 数	意 义	“最小”值
DBL_MAX	最大的双精度数	1E+37
LDBL_MAX	最大的长双精度数	1E+37
FLT_EPSILON	单精度数的机器误差 ϵ	1E-9
DBL_EPSILON	双精度数的机器误差 ϵ	1E-9
LDBL_EPSILON	长双精度数的机器误差 ϵ	1E-9

* FLT_ROUNDS 的值:

- 1 不能确定
- 0 趋向 0
- 1 最接近的数
- 2 趋向正无穷大
- 3 趋向负无穷大

6.2 <math.h>

尽管 C 在科学计算编程中的应用不是很广泛，但是它却提供了一组具有丰富特征的数学函数（见表 6.2）。这些函数中大多数函数的名字是在数学中使用的标准名字，因此，只要知道它们的名字，也就知道了如何使用它们。然而，还有一些函数值得特殊说明，如在程序清单 6.1 中，函数 fmod 计算了第一个参数除以第二个参数所得到的余数，结果是 62.9558，因为

$$1234.56 - 13 * 90.1234 = 62.9558。$$

表 6.2 <math.h> 中定义的函数

acos	反余弦
asin	反正弦
atan	反正切（主值）
atan2	反正切（整圆值）
ceil	上限
cos	余弦
cosh	双曲余弦
exp	e 次幂

<code>fabs</code>	绝对值
<code>floor</code>	取整（函数中最大的整数）
<code>fmod</code>	取模（余数）
<code>frexp</code>	标准化小数指数部分
<code>ldexp</code>	与 <code>frexp</code> 相反
<code>log</code>	自然对数
<code>log10</code>	以 10 为底的对数
<code>modf</code>	取整/小数部分
<code>pow</code>	将数转换为幂的形式
<code>sin</code>	正弦
<code>sinh</code>	双曲正弦
<code>sqrt</code>	平方根
<code>tan</code>	正切
<code>tanh</code>	双曲正切

程序清单 6.1 说明 <math.h> 中几个函数

```

#include <stdio.h>
#include <math.h>

main()
{
    double x = 1234.56, y = 90.1234, z, w;
    int p;

    printf("x == %g, y == %g\n", x, y);
    printf("fmod(x,y) == %g\n", fmod(x,y));
    printf("floor(y) == %g\n", floor(y));
    printf("ceil(y) == %g\n", ceil(y));
    w = modf(y,&z);
    printf("after modf(y,&z): w == %g, z == %g\n", w, z);
    w = frexp(y,&p);
    printf("after frexp(y,&p): w == %g, p == %d\n", w, p);
    printf("ldexp(w,p) == %g\n", ldexp(w,p));
    return 0;
}

```

```

/* 输出 */
x == 1234.56, y == 90.1234
fmod(x,y) == 62.9558
floor(y) == 90
ceil(y) == 91
after modf(y,&z): w == 0.1234, z == 90
after frexp(y,&p): w == 0.704089, p == 7
ldexp(w,p) == 90.1234

```

函数 `modf` 将其参数的整数部分放到由第二个参数所指向的位置，并返回小数部分。函数 `frexp` 计算一个标准化小数 x 和一个整数 p ，因此它的参数相当于：

$$x \cdot 2^p$$

它的逆运算 `ldexp` 计算已知 x 和 p 所得的浮点数。

如果一个数是整数，那么它的最大整数部分就是它本身，否则就是这个数在数轴上“左”边相邻的数，因此以下的关系式是有效的：

```

floor(90.1234)==90
floor(-90.1234)==-91

```

一个数的上限就是它在数轴上右边相邻的数，因此：

```

ceil(90.1234)==91
ceil(-90.1234)==-90

```

程序清单 6.2 中的计算器程序也说明了 `<math.h>` 中的一些函数。

程序清单 6.2 一个说明了一些 `<math.h>` 函数的简单的计算器程序

```

/* calc.c: 低智能计算器 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <ctype.h>

#define LINSIZ 40

char *getline(char *);
main()
{
    double reg = 0.0;
    char line[LINSIZ];

    while (getline(line) != NULL)
    {
        char *op;
        double val;

```

```
        /* 解析命令字符串 */
        val = strtod(line,&op);
        while (isspace(*op))
            ++op;
        strupr(op);

        /* 执行操作 */
        if (*op == '+')
            reg += val;
        else if (*op == '-')
            reg -= val;
        else if (*op == '*')
            reg *= val;
        else if (*op == '/')
        {
            if (val != 0)
                reg /= val;
            else
            {
                puts("ERROR> invalid divisor <");
                continue;
            }
        }
        else if (*op == '=')
            reg = val;
        else if (*op == '^')
        {
            if (val < 0.0)
            {
                puts("ERROR> invalid exponent <");
                continue;
            }
            else if (val == 0.0)
                reg = 1.0;
            else if (val == 0.5)
                reg = sqrt(reg);

            else
                reg = pow(reg,val);
        }
        else if (strncmp(op,"NEGATE",1) == 0)
            reg = -reg;
        else if (strncmp(op,"MOD",1) == 0)
        {
            if (val == 0.0)
```

```

        {
            puts("ERROR> invalid modulus <");
            continue;
        }
        else
            reg = fmod(reg, val);
    }
    else if (strncmp(op, "CEIL", 1) == 0)
        reg = ceil(reg);
    else if (strncmp(op, "FLOOR", 1) == 0)
        reg = floor(reg);
    else if (strncmp(op, "ROUND", 1) == 0)
        reg = (reg < 0.0) ? ceil(reg - 0.5)
                        : floor(reg + 0.5);
    else if (strncmp(op, "SQRT", 1) == 0)
        reg = sqrt(reg);
    else if (strncmp(op, "QUIT", 1) == 0)
        exit(0);
    else if (*op != '\0')
    {
        puts("ERROR> invalid operation <");
        continue;
    }

    printf("\t%s = %g\n", line, reg);
}
return 0;
}

char *getline(char *buf)
{
    fputs("Calc> ", stdout);
    fflush(stdout);
    return gets(buf);
}

```

6.3 <errno.h>

头文件<errno.h>实现了一个简单的错误报告功能。它定义了一个全局整数 `errno` 来保存由大量的库函数所产生的错误代码。C 标准需要销售商提供仅仅两个代码，即 `EDOM` 和 `ERANGE`（见表 6.3）。代码 `EDOM` 表示了一个区域错误，这通常意味着向函数传递了错误的参数。例如，<math.h> 中的 `sqrt` 函数在当求一个负数的平方根的时候会发出错误信息。有些数学函数把 `errno` 置为 `ERANGE` 来确认范围错误代码，这意味着对有效参数的计算可能会产

生数学上的上溢或者下溢。标准库中的大多数数学函数使用 `errno` 来报告这种错误。如程序清单 6.2 中的计算器程序所示，在调用任何使用这个功能的函数之前应该把 `errno` 置为 0，并且在函数返回后立刻检查。函数 `perror` 打印了其后有一个冒号的字符串参数，再其后是 `errno` 中最近的一次错误记录。表达式 `perror(s)` 相当于以下的表达式：

```
printf("%s: %s\n", s, strerror(errno));
```

<string.h>中定义的函数 `strerror` 返回 `errno` 相应的文本。

表 6.3

<errno.h>中的定义

errno	全局整数表示错误
EDOM	区域错误代码
ERANGE	范围错误代码

下面来自于其他的标准库头文件中的函数也将 `errno` 置为失败, 它们是: `strtd`, `strol`, `strtoul`, `fgetpos`, `fsetpos` 和 `signal`。一个实现可以自由地提供除 `EDOM` 和 `ERAGNE` 之外其他的错误代码, 并且和其他函数一起使用 `errno` 功能, 但是这种用法显然是不可移植的。

6.4 <locale.h>

标准 C 中的本地化是一个处理的优先级和显示对文化、语言、或民族起源敏感的信息的集合，例如日期和货币的格式。共有 5 类这样的信息，它们由 `<locale.h>` 中定义的宏来命名，在这 5 类中本地化起作用（见表 6.4）。每类都可被设为不同的本地化（例如“american”、“italian”，等等）。为了得到一个更好的术语，我把对所有类别设置的集合称作本地化轮廓。

表 6.4

本地化分类

种 类	功 能
LC_COLLATE	使 <code>strcoll</code> 、 <code>strxfrm</code> 、 <code>wscoll</code> 和 <code>wcsxfrm</code> 适应于本地化的语言文化
LC_CTYPE	使 <code>ctype.h</code> 中的 <code>isxxx/iswxxx</code> 函数适应与本地化有关的字符集，并影响多字符和宽字符映射
LC_MONETARY	设置参数保持货币值的显示，如小数点、分组（例如以千分组）、组分隔符等。这纯属建议，并不影响标准库函数
LC_NUMERIC	类似 <code>LC_MONETARY</code> ，但用于非货币值（例如小数点可能与非货币值不同）
LC_TIME	使 <code>strftime</code> 和 <code>wcsftime</code> 适应文化规范

标准 C 规范了用于直接处理本地化的两个函数:

```
struct lconv *localeconv(void);
```

```
char *setlocale(int category, char *locale);
```

localeconv 结构体中的成员如表 6.5 所示。函数 localeconv 返回一个静态的 localeconv 对象，这个对象

包含了对 LC_MONETARY 和 LC_NUMERIC 两种类的设置，setlocale 把已知种类的本地化变为在本地化中所述的那样。可以通过指定一个 LC_ALL 类（见程序清单 6.3）把所有的种类都设置为已知的本地化。如果本地化为 NULL，则返回种类的当前本地化字符串。所有的实现都必须支持最小化的“C”本地化，以及由空字符串命名的本地化（可以和“C”本地化相同）。

表 6.5

lconv 结构体中的成员

```

struct lconv
{
    char *decimal_point;
    char *thousands_sep;
    char *grouping;
    char *int_curr_symbol;
    char *currency_symbol;
    char *mon_decimal_point;
    char *mon_thousands_sep;
    char *mon_grouping;
    char *positive_sign;
    char *negative_sign;
    char int_frac_digits;
    char frac_digits;
    char p_cs_precedes;
    char p_sep_by_space;
    char n_cs_precedes;
    char n_sep_by_space;
    char p_sign_posn;
    char n_sign_posn;
};

```

程序清单 6.3 说明本地化设置对小数点字符和时间格式的影响

```

/* tlocale.c: 举例说明 setlocale() -
 *
 * 注意：
 * 在 Windows NT 下编译 Visual C++

```

```
*/

#include <locale.h>
#include <stdio.h>
#include <time.h>

void print_stuff(void);

main()
{
    /* 首先在 C 环境下 */
    puts("In the C locale:");
    print_stuff();

    /* 现在尝试在 German 环境下 */
    puts("\nIn the German locale:");
    setlocale(LC_ALL, "german");
    print_stuff();
    return 0;
}

void print_stuff(void)
{
    char text[81];
    time_t timer = time(NULL);

    printf("%.2f\n", 1.2);
    strftime(text, sizeof text, "%A, %B %d, %Y (%x)\n",
              localtime(&timer));
    puts(text);
}

/* 输出 */
In the C locale:
1.20
Monday, March 17, 1997 (03/17/97)

In the German locale:
1,20
Montag, Mrz 17, 1997 (17.03.97)
```

6.5 <setjmp.h>

当在一组嵌套函数调用中遇到异常条件时，需要一个超级转向（“super goto”）来跳转到

函数调用堆栈中一个安全的上级位置。这就是 `setjmp/longjmp` 机制的作用。用户记录了一个返回到带有 `setjmp` 的位置，并通过 `longjmp` 跳转到这个位置。下面是相应的语法：

```
#include<setjmp.h>
jmp_buf recover;
main()
{
    volatile int i=0;
    for(;;)
    {
        if(setjmp(recover)!=0)
        {
            /* Recover from error in f() */
        }
        /* Get deeply nested... */
    }
    return 0;
}
...
void f()
{
    /* Do some risky stuff */
    if (<things go crazy>)
        longjmp(recover,1);
    /* else carry on */
}
```

`jmp_buf` 是一个数组，该数组用来保存必要的系统信息以恢复在 `setjmp` 位置的执行。很显然，跳转缓冲区一定是全局的。当调用 `setjmp` 时，系统在 `recover` 中存储了调用环境参数，如堆栈和指令指针寄存器的内容。当直接调用时，`setjmp` 总是返回 0。调用 `longjmp` 可恢复调用环境，因此执行回到 `setjmp` 调用的地方继续进行，但是有一点不同：似乎 `setjmp` 已经从 `longjmp` 的调用返回了第二个参数（这里是 1）一样。如果把 `longjmp` 第二个参数置为 0，则无论如何都返回 1。由于 `longjmp` 完成了从一个函数的交替返回，它会中断程序的正常流程，因此只应该用它来处理不寻常的条件。关于 `setjmp/longjmp` 更全面的处理机制请参见第 12 章。

6.6 <signal.h>

当一个异常事件中断程序的正常运行时，比如除数为 0 的错误或者当用户按下功能键时（例如 `Control-C` 或 `Del`），信号（signed）就会出现。头文件 `<signal.h>` 定义了六个“标准信号”，如表 6.6 所示。这些信号源于 UNIX 下的 PDP 结构，并非所有的信号都可以应用于系统。实现也可以定义其他的信号，或者它可能完全忽略信号，因此处理信号自然是不可移植的。第

12 章详细探讨了信号处理。

表 6.6

标准信号

SIGABRT	非正常结束（由 abort()发出）
SIGFPE	计算异常（例如溢出）
SIGILL	无效的函数映像（例如，非法指令）
SIGINT	交互式部署（例如 Control-C）
SIGSEGV	试图访问受保护内存
SIGTERM	终止请求

6.7 <stdarg.h>

这个头文件提供了一个定义带有可变长度参数列表的函数的功能，就像 printf 那样。从编译器的<stdio.h>的 printf 的函数原型看起来应该如下：

```
int printf(const char*, ...);
```

省略号告诉编译器，在 printf 的调用中，在第一个参数的后面允许没有参数，或有更多的任意类型的参数。为使 printf 正确地执行，格式化字符串后面的参数必须与格式化字符串中对应的编辑描述符匹配。如果参数的个数比格式化字符串所期望的少，那么结果是不确定的。如果参数多，它们就会被忽略。总之，在一个函数的原型中使用省略号时，就告诉编译器不需要对可选参数进行类型检查。（<stdarg.h>中定义的列表，见表 6.7）。

表 6.7

<starg.h>中的定义

类型	描述
va_list	一个可变长度的参数列表
宏	描述
va_start	初始化 va-list
va_arg	得到 va-list 中的下一个参数
va_end	关闭 va-list

程序清单 6.4 中的程序说明了如何使用 va_list 机制在一个可变长度参数列表中找到最大的整数。正如你所见到的，使用可变长度参数列表需要做两件事：

1. 至少有一个固定的参数（总是省略号前的最后一个参数）来初始化 va_list，并且
2. 一些能够将参数的个数或类型传达给函数的机制。

因此，如果能够满足上述前提的话，下面的函数原型就没有用了：

```
void f(..);                /*未知参数的位置*/
```

程序清单 6.4 用<stdarg.h>宏来查找整数的可变长度列表

```
/* max.c */
#include <stdio.h>
#include <stdarg.h>

int maxn(size_t count, ...)
{
    int n, big;
    va_list numbers;

    va_start(numbers, count);

    big = va_arg(numbers, int);
    while (count--)
    {
        n = va_arg(numbers, int);
        if (n > big)
            big = n;
    }

    va_end(numbers);
    return big;
}

main()
{
    printf("max = %d\n", maxn(3, 1, 3, 2));
    return 0;
}

/* 输出 */
max = 3
```

有很多方法可以满足第二个要求。程序清单 6.5 中的程序把可变个数的字符串和它固定的字符串参数连接在一起。这个程序处理了一个又一个字符串，直到它在 `va_list` 中找到了 `NULL` 指针为止。可调用

```
concat(s, NULL);
```

把 `s` 初始化为空字符串。

程序清单 6.5 连接一些可变的字符串

```
/* concat.c */
```

```

#include <stdarg.h>
#include <stdio.h>
#include <string.h>
char * concat(char *s,...)
{
    va_list strings;
    char *p;
    /* 复制第一个字符串 */
    va_start(strings,s);
    if ((p = va_arg(strings,char *)) == NULL)
    {
        *s = '\0';
        return s;
    }
    else
        strcpy(s,p);

    /*添加其他的 */
    while ((p = va_arg(strings,char *)) != NULL)
        strcat(s,p);
    return s;
}

main()
{
    char buf[128];
    concat(buf,"Sweet","Talker","Betty","Crocker",NULL);
    printf("\n%s\n",buf);
    return 0;
}

/* 输出 */
"SweetTalkerBettyCrocker"

```

6.8 va_list 作为参数

程序清单 6.6 中有一个很有用的函数 `fatal`，该函数恰如其分地向 `stderr` 打印一个格式化的信息然后退出。用一个格式化的字符串和一个参数列表，可以像使用 `printf` 一样调用它。例如：

```
fatal("Error %d on device %d\n",err,dev);
```

你可能愿意做的就是将格式化字符串和打印参数传递给实现 `printf` 机制的函数。这些函数实现了。C 库函数 `vfprintf` 使这件事变得非常容易。你所要做的就是用打印参数来初始化 `va_list` 并且把它作为第三个参数传递。正像你预期的那样，C 库也有伙伴式的函数 `vprintf` 和 `vsprintf`。

使用其他语言的程序员也许会很奇怪为什么需要所有这些机制。例如 Fortran 程序员非常

习惯于打印一些在参数列表中没有显式给出关于参数的数量或类型信息的语句：

```
*   Output two numbers:
PRINT *,x,y
```

要找到数据列表中的最大值，只会有以下的数据出现：

```
PRINT*,MAX()
```

程序清单 6.6 通过把 `va_list` 传递给 `vfprintf` 来创建变量格式字符串

```
/* fatal.c: 退出带错误信息的程序
#include<stdio.h>
#include<stdlib.h>
#include<stdarg.h>
#include<string.h>

void fatal(char *fmt,...)
{
    va_list args;
    if(strlen(fmt)>0)
    {
        va_start(args,fmt);
        vfprintf(stderr,fmt,args);
        va_end(args);
    }
    exit(1);
}
```

能在 Fortran 中这样做的原因是如 `PRINT` 和 `MAX` 语句是语言的一部分（Fortran 把他们称作固有函数）。编译器知道它们的要求因此可以提供合适的信息。换句话说，在 C 中，除了操作者所提供的功能之外，没有输入、输出或者其他任何构建在语言内部的功能。C 的哲学就是让语言更简练，并用库来提供所需的功能。由于库和编译器之间惟一的通信是函数调用机制，所以调用函数时就必须提供所需的信息。

6.9 应用

在金融和其他的数据应用中常常需要表示以逗号分开的整数，例如货币金额：

```
$11,235,852
```

一种把数字转换成字符串的方法是先使用 `sprintf`，然后再将字符串回传，把它拷贝到另一个字符串中，并在需要的位置上插入逗号。另一种就我要在此给出的方法解决了创建字符串回传时的常见问题。

程序清单 6.7 使用函数 `prepend` 来构建一个字符串回传。你把 `pffset` 传至输出缓冲，`pffset` 指向已连接字符串的第一个字符，然后把一个待连接的字符串传给 `prepend`，返回值是新的 `pffset`。

程序清单 6.7 说明 prepend 的用法

```

#include <stdio.h>
#include <assert.h>
#define WIDTH 11
extern int prepend(char *, unsigned, char *);

main()
{
    char s[WIDTH+1];
    int offset = WIDTH;

    s[offset] = '\0';
    offset = prepend(s, offset, "three");
    assert(offset >= 0);
    puts(s+offset);

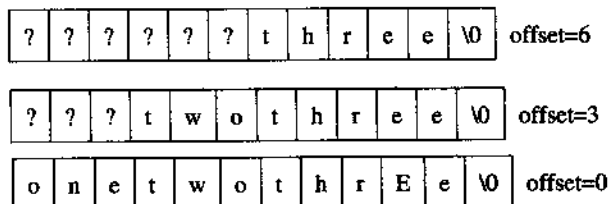
    offset = prepend(s, offset, "two");
    assert(offset >= 0);
    puts(s+offset);

    offset = prepend(s, offset, "one");
    assert(offset >= 0);
    puts(s+offset);
    return 0;
}

/* 输出 */
three
twothree
onetwothree

```

下面的图表显示了每次调用 `prepend` 后 `s[]` 的状态：



程序清单 6.8 中有 `prepend` 和另一个函数 `preprintf` 的实现，后者允许预先把字符串格式化。函数 `preprintf` 使用 `vsprintf` 创建一个格式化字符串，然后调用 `prepend`，把这个字符串加到已存在的字符串的前面。

程序清单 6.8 建立回传字符串的函数

```

/* preprint.c: 预先安排字符串的函数 */
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <stdlib.h>

int prepend(char *buf, unsigned offset, char *new_str)
{
    int new_len = strlen(new_str);
    int new_start = offset - new_len;

    /* 将字符串压栈到另一个的前面 */
    if (new_start >= 0)
        memcpy(buf+new_start, new_str, new_len);

    /* 返回新的开始位置(如果溢出则为负数) */
    return new_start;
}

int preprintf(char *buf, unsigned offset, char *format, ...)
{
    int pos = offset;
    char *temp = malloc(BUFSIZ);

    /* 格式化, 然后压栈 */
    if (temp)
    {
        va_list args;

        va_start(args, format);
        vsprintf(temp, format, args);
        pos = prepend(buf, offset, temp);
        va_end(args);
        free(temp);
    }
    return pos;
}

```

现在, 我可以根据 `prepend` 和 `preprintf` 来实现一个函数 `commas` (见程序清单 6.9)。通过普通的求余和求商运算, 依次从右向左选取数字, 然后把这个数字放到一个静态的字符缓冲中, 并在需要的地方插入逗号。函数 `commas` 返回一个指向完整字符串起始地址的指针, 这个指针可能和缓冲的起始位置一致, 也可能不一致。注意数字的基数和这组数的大小是参数化了的。

程序清单 6.9 用 prepend 和 preprintf 格式化具有逗号分割符的数字

```

/* commas.c: 将数字转换成具有逗号分割符的字符串 */
#include <stdio.h>

#define BASE 10
#define GROUP 3

/* 需要空间来保存无符号长整型数位数,
 * 插入逗号和空字节, 这取决于上面的
 * BASE 和 GROUP (但从对数算法上看, 不能,
 * 作为一个常数, 因而我们必须在这里人为地定义它)
 */
#define MAXTEXT 14      /* BASE = 10 */

int prepend(char *, unsigned, char *);
int preprintf(char *, unsigned, char *, ...);

char *commas(unsigned long amount)
{
    short offset = MAXTEXT-1, /*字符串“开始”处 */
          place;              /*当前数字 BASE 的幂*/
    static char text[MAXTEXT];

    text[offset] = '\0';

    /*用逗号从右向左压栈数字*/
    for (place = 0; amount > 0; ++place)
    {
        if (place % GROUP == 0 && place > 0)
            offset = prepend(text, offset, ",");
        offset = preprintf(text, offset, "%x", amount % BASE);
        amount /= BASE;
    }

    return (offset >= 0) ? text + offset : NULL;
}

main()
{
    puts(commas(1));
    puts(commas(12));
    puts(commas(123));
    puts(commas(1234));
    puts(commas(12345));
    puts(commas(123456));
    puts(commas(1234567));
}

```

```

    puts(commas(12345678));
    puts(commas(123456789));
    puts(commas(1234567890));
    return 0;
}

/* 输出 */
1
12
123
1,234
12,345
123,456
1,234,567
12,345,678
123,456,789
1,234,567,890

```

6.10 结论

在第4、5和6章中我已经尽量给出了标准C库的功能特点。自己重新建立这些功能是很愚蠢的。虽然我已经在三组依次递减的优先级中说明了库，但是我的优先级可能和你自己的不相符合。对你来说掌握整个库是很明智的。

6.11 浮点数系统

长久以来，实数的计算机数学运算是使学生和与之相似的从业者产生挫折感和困惑的原因。程序清单6.10中的程序说明了简单的序列求和是怎样出错的。它用下面的公式来计算表达式 e^x ：

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

程序清单 6.10 说明计算 e 的 x 次幂时的舍入误差

```

/* round.c */
#include <stdio.h>
#include <math.h>

double e(double x);

```

```

main()
{
    printf("e(55.5) == %g, exp(55.5) == %g\n",
           e(55.5), exp(55.5));
    printf("e(-55.5) == %g, exp(-55.5) == %g\n",
           e(-55.5), exp(-55.5));
    printf("1/e(55.5) == %g\n", 1.0 / e(55.5));
    return 0;
}

double e(double x)
{
    double sum1 = 1.0;
    double sum2 = 1.0 + x;
    double term = x;
    int i = 1;

    /*通过泰勒级数计算 exp(x) */
    while (sum1 != sum2)
    {
        sum1 = sum2;
        term = term * x / ++i;
        sum2 += term;
    }
    return sum2;
}

/* 输出 */
e(55.5) == 1.26866e+24, exp(55.5) == 1.26866e+24
e(-55.5) == -3.92623e+06, exp(-55.5) == 7.88236e-25
1/e(55.5) == 7.88236e-25

```

当参数是正数时程序一切工作正常，但是参数是负数时的结果南辕北辙。当然，问题是计算机的能力有限，它只能表示实数集中一个微小的子集。有很多数字分析是用来处理舍入误差的，这是有限精度计算中最危险的事情。

随着时间的流逝，许多计算机使用了定点数字系统，在这里数字从根值 (b), 精度 (p), 和小数大小 (f) 被导出。例如, 值 b=10, p=4 和 f=1 定义了下面的 19,999 个数字的集合:

$F = \{-999.9, -999.8, \dots, 999.8, 999.9\}$

既然这些数字是均匀分布的，这个系统中表示任何实数 x 时的最大绝对误差被限制在 0.5 之内，换句话说也就是

$$|x - \text{fix}(x)| \leq 0.5$$

机器整数集构成了 f=0 的定点系统，0.5 左右的系统最大绝对误差和截断误差不大于 1.0 的系统。

然而，在数学计算中绝对误差并不是普遍有用的。在大多数情况下，你更感兴趣的是百

分率或者是一个数字相对于另一个的差别。可以用下列公式计算 y 相对于 x 的相对误差：

$$\frac{|x - y|}{|x|}$$

考虑一下数字 865.54 和 .86554 在 F 中是如何表示的：

```
fix(865.54)=865.5
```

```
fix(.86554)=.9
```

因为十进制数的个数是固定的，第二个数比第一个数更不合适，这由相对误差可看出：

$$\text{rel}(865.54) = \frac{|865.54 - 865.5|}{865.54} = .0000462$$

$$\text{rel}(.86554) = \frac{|.86554 - .9|}{.86554} = .0398$$

第二个要比第一个糟糕 1000 倍！

如今计算机提供了浮点数字系统，它用下面的形式来表示数字：

$$\pm 0.d_1d_2\dots d_p \times \beta^e$$

这里 $m \leq e \leq M, 0 \leq d_i < \beta$ 。

除了基部 (β) 可以不是 10，且精度 (p) 有上限以外，这种表示方法与在学校里所学的科学记数法一样。大多数浮点系统使用标准化的表示，这意味着 d_1 不能是 0。这些数字系统被称做浮点，这种叫法是有明显的理由——根值点“浮动”加上指数 e 的调节保证了正确的值。现在考虑一个由 $\beta=10, p=4, m=-2$ 和 $M=3$ 定义的浮点系统 G 。前面所用过的数字可表示为：

```
f1(.86554) = .8655 × 100
```

```
f1(865.54) = .8655 × 103
```

现在看一下在计算这两个表示的相对误差时发生了什么情况：

$$\text{rel}(865.54) = \frac{|865.54 - 865.5|}{865.54} = .0000462$$

$$\text{rel}(.86554) = \frac{|.86554 - .8655|}{.86554} = .0000462$$

在一个浮点系统中，无论数字和幂是什么，相对误差都相同。可以看出在一个浮点系统范围之内所能表示的任何实数的相对误差都不大于 β^{1-p} ，这个数在 G 中是 0.001。

浮点数字不是均匀分布的。例如，在 G 中下一个比 1 大的数是 1.001，它比 1 多了 .001，但下一个 $(\beta^{-1}) \beta^{p-1}$ 比 10 大的数是 10.01，它比 10 多了 .01。通常，在 β 的幂数所表示的数中如两个数 β^e 和 β^{e+1} 之间的间距是 β^{e+1-p} 。一个微不足道的组合分析表明每个区间 $[\beta^e, \beta^{e+1}]$ 都有相同的浮点数字—— $(\beta^{-1}) \beta^{p-1}$ ，并且这些浮点数字的总和是 $2(M-m+1)(\beta-1)\beta^{p-1}+1$ （因此 p 增加时这个数字系统密度也随之增加）。如上所示，虽然值小的数之间比较稠密，值大的数之间比较分散，但是在整个系统中连续的浮点数之间的相对间距实质上是相同的。特别地，在区间 $[\beta^e, \beta^{e+1}]$ 以内的 2 个邻近数字之间的相对间距是 β^{1-p} ，并且在 β^{e+1} 和与其前面紧密

相邻的数之间的相对间隔是 β^p 。

β^{1-p} 恰巧等于 1.0 和它后面紧跟着的数之间的间隔，这个间距被称为机器误差 ε ，它是浮点数字系统间隔的很好的检测尺度。除零以外最小的浮点数当然是 β^{m-1} ，它通常表示为 σ ，而最大数 λ 则是 $\beta^M (1-\beta^p)$ 。

标准 C 头文件 `<float.h>` 为下列所有参数提供常量：

```

β==FLT_RADIX
ρ==DBL_MANT_DIG
m==DBL_MIN_EXP
M==DBL_MAX_EXP
ε==DBL_EPSILON
σ==DBL_MIN
λ==DBL_MAX

```

头文件 `<float.h>` 也给出了最后 5 个参数的浮点和长双精度等价形式，以表示在标准 C 中的 3 个（没有必要不同）浮点数字系统（见表 6.1）。

程序清单 6.11 中的程序用中二分法来计算 x^2+x+1 在区间 $[-1, 1]$ 的根。此算法将区间（此区间必须包含函数 f 的符号变化）对分，直到端点的相对间距小于或等于机器误差 ε 。如果要想得到比这更大的精度的话，这个循环将永不终止。

程序清单 6.11 举例说明求根算法中机器误差 ε 的应用

```

/* root.c */
#include <stdio.h>
#include <float.h>
#include <math.h>
#include <assert.h>

#define sign(x) ((x < 0.0) ? -1 : (x > 0.0) ? 1 : 0)

#define PREC DBL_DIG
#define EPS DBL_EPSILON

typedef double ftype;

ftype root(ftype a, ftype b, ftype (*f)(ftype))
{
    ftype fofa = f(a);
    ftype fofb = f(b);
    assert(a < b);
    assert(fofa * fofb < 0.0);

    /*用二分法的方式接近根*/
    while (fabs(b - a) > EPS*fabs(a))

```

```

{
    ftype x = a + (b-a)/2.0;
    ftype fofx = f(x);
    if (x <= a || x >= b || fofx == 0.0)
        return x;
    if (sign(fofx) == sign(fofa))
    {
        a = x;
        fofa = fofx;
    }
    else
    {
        b = x;
        fofb = fofx;
    }
}
return a;
}

main()
{
    extern ftype f(ftype);
    printf("root == %.*f\n",PREC,root(-1.0,1.0,f));
    return 0;
}

ftype f(ftype x)
{
    return x*x + x - 1.0;
}

/* 输出 */
root == 0.618033988749895

```

如果你发现自己在一个不提供<float.h>中参数的环境中，就可以直接计算 β 、 ρ 和 ε 。为了弄清楚这是如何进行的，记住在连续的浮点数字之间的间距随数的大小而增加。最终会发现这样一个点，在这点上相邻数之间的间距比 1 大。实际上第一个满足条件的区间是 $[\beta^p, \beta^{p+1}]$ ，因为区间内的整数在它们的表达式中要求 $p+1$ 位。在一个浮点系统中正整数精确地说是下列这些数：

$$1, 2, \dots, \beta^{p-1}, \beta^p, \beta^p + \beta, \beta^p + 2\beta, \dots, \beta^{p+1}, \beta^{p+1} + \beta^2, \dots$$

为了得到根值，程序清单 6.12 中 a 不停地加倍，直到到达间距超过 1 的一个点为止。然后找到下一个更大的浮点数字并且减去 a 得到 β （程序清单 6.12 中为 b ）。接着程序计算 b 的最小值，以再一次得到相邻数之间间距大于 1 的点，这个点的幂就是精度的 p 。为了得到 ε ，程序找到紧挨着 1 右边的数字，并用这个数减去 1。

程序清单 6.12 计算机器浮点参数

```
/* machine.c */
#include <stdio.h>
#include <math.h>
#include <float.h>

main()
{
    int beta, p;
    double a, b, eps, epspl, sigma, nums;

    /* 发现根 */
    a = 1.0;
    do
    {
        a = 2.0 * a;
        b = a + 1.0;
    } while ((b - a) == 1.0);

    b = 1.0;
    do
    {
        b = 2.0 * b;
        while ((a + b) == a);
        beta = (int) ((a + b) - a);
        printf("radix:\n");
        printf("\talgorithm: %d\n", beta);
        printf("\tprovided: %d\n", FLT_RADIX);

        /* 计算精确到位 */
        p = 0;
        a = 1.0;
        do
        {
            ++p;
            a *= (double) beta;
            b = a + 1.0;
        } while ((b - a) == 1.0);
        printf("precision:\n");
        printf("\talgorithm: %d\n", p);
        printf("\tprovided: %d\n", DBL_MANT_DIG);

        /* 计算机器误差 */
        eps = 1.0;
        do
        {
```



```

        eps = 0.5 * eps;
        epspl = eps + 1.0;
    } while (epspl > 1.0);
    epspl = 2.0 * eps + 1.0;
    eps = epspl - 1.0;
    printf("machine epsilon:\n");
    printf("\talgorithm: %g\n",eps);
    printf("\tformula: %g\n",pow(FLT_RADIX,1.0-DBL_MANT_DIG));
    printf("\tprovided: %g\n",DBL_EPSILON);

/* 计算最小标准化幅值 */
    printf("smallest nonzero magnitude:\n");
    printf("\tformula: %g\n",pow(FLT_RADIX,DBL_MIN_EXP-1));
    printf("\tprovided: %g\n",DBL_MIN);

/* 计算最大标准化幅值 */
    printf("largest nonzero magnitude:\n");
    printf("\tformula: %g\n",

        pow(FLT_RADIX,DBL_MAX_EXP-1) * FLT_RADIX *
        (1.0 - pow(FLT_RADIX,-DBL_MANT_DIG)));
    printf("\tprovided: %g\n",DBL_MAX);

    printf("smallest exponent: %d\n",DBL_MIN_EXP);
    printf("largest exponent: %d\n",DBL_MAX_EXP);

    nums = 2 * (FLT_RADIX - 1)
        * pow(FLT_RADIX,DBL_MANT_DIG-1)
        * (DBL_MAX_EXP - DBL_MIN_EXP + 1);
    printf("This system has %g numbers\n",nums);
    return 0;
}

/*输出 (MS Visual C++ 4.2) */
radix:
    algorithm: 2
    provided: 2
precision:
    algorithm: 53
    provided: 53
machine epsilon:
    algorithm: 2.22045e-016
    formula: 2.22045e-016
    provided: 2.22045e-016
smallest nonzero magnitude:
    formula: 2.22507e-308

```

```
provided: 2.22507e-308
largest nonzero magnitude:
  formula: 1.79769e+308
  provided: 1.79769e+308
smallest exponent: -1021
largest exponent: 1024
This system has 1.84287e+019 numbers
```

第二部分

主要概念

抽 象

不同于机器，人脑对处理复杂事情的数量有严格的限制。生活本来就是复杂的，模仿真实的软件系统也一样。为了掌握一个复杂系统，我们只把精力集中在一些已给出上下文意思的最主要的问题上，而忽略其他的问题，这是很有必要的。这个过程叫做抽象，它可以使系统设计者用有组织的、方便的方法去解决一些复杂的问题。就像 Grady Booch 说的那样，“相对于阅读者的观点而言，一个抽象表示一个对象区别于其他对象的本质特征，因此它能够提供清楚定义了的、概念上的界限。”（*Object-Oriented Analysis and Design with Applications*，第2版，Benjamin-Cummings，1994，p.41）。

7.1 数据抽象

在软件中抽象经常通过用户自定义数据类型来表示，对象的新类是由系统预定义的类型或者其他用户自定义类型组成的。当然，这不是新概念，但很好地支持了它的语言，如 Ada，C++，CLOS，Java，Eiffel，以及 Smalltalk，只是最近才得到广泛的应用。

如果你已经用 C 语言进行了一段时间的编程，可能已经用到了结构体（struct）机制。你也不得不提供处理这些结构体的函数。不论你知道与否，你正在模拟一个用户自定义类型。举个例子，碰巧你需要在一个程序里处理日期。程序清单 7.1 定义了一个 Date 结构体和两个处理日期的函数，其中一个函数是把时间格式化成年月日的形式，而另一个函数进行日期的比较。（函数的实现见程序清单 7.2，测试程序见程序清单 7.3）。

为了使用这个抽象数据类型，你只需要知道这两个函数的协议，换句话说，只须知道它的接口。你并不需要知道函数是怎么被执行的，也不需要知道结构体的构造。一个定义明确的抽象允许你只注意其外型，而忽略实现细节。在接口和实现之间设置障碍的概念称为封装。当然，对 Date 类型的封装障碍有一个漏洞，因为可以绕过接口直接操纵一个结构体成员，如

```
dp->month = 7;
```

程序清单 7.1 C 形式的日期 (Date) 类型

```
/* date.h */
#ifndef DATE_H
#define DATE_H

struct Date
{
    int month;
    int day;
    int year;
};

typedef struct Date Date; /*在 C++ 中不需要 */

char *date_format(const Date*, char*);
int date_compare(const Date*, const Date*);

#endif
```

程序清单 7.2 Date 类型的实现

```
/* date.c */
#include <stdio.h>
#include "date.h"

static const char *month_text[] =
{
    "Bad month", "January", "February", "March", "April",
    "May", "June", "July", "August", "September",
    "October", "November", "December"
};

char* date_format(const Date* dp, char* buf)
{
    sprintf(buf, "%s %d, %d",
            month_text[dp->month], dp->day, dp->year);
    return buf;
}

int date_compare(const Date* dp1, const Date* dp2)
{
    int result = dp1->year - dp2->year;
    if (result == 0)
        result = dp1->month - dp2->month;
    if (result == 0)
        result = dp1->day - dp2->day;
}
```

```
    return result;
}
```

程序清单 7.3 测试 Date 类型

```
/* tdate.c */
#include <stdio.h>
#include "date.h"

#define DATELEN 19

main()
{
    Date d1 = {10,1,1951}, d2 = {3,7,1995};
    char buf[DATELEN+1];
    int cmp;

    printf("d1 == %s\n",date_format(&d1,buf));
    printf("d2 == %s\n",date_format(&d2,buf));
    cmp = date_compare(&d1,&d2);
    printf("d1 %s d2\n", cmp < 0 ? "precedes"
                               : (cmp > 0) ? "follows"
                               : "equals");

    return 0;
}

/* 输出:
d1 == October 1, 1951
d2 == March 7, 1995
d1 precedes d2
*/
```

为了弥补这个漏洞,可以利用接口本身不需要知道关于 Date 的任何信息这个事实,它只需要一个指向 Date 类型的指针。在程序清单 7.4 中,我把 Date 定义为一个不完全类型,我仅仅声明了它的存在,而没有做别的事情。作为一个用户看到的只是 data2.h,而对于 Date 的构造或实现却一无所知。我在程序清单 7.5 中完成了这个类型(测试程序在程序清单 7.6 中)。然而,这个额外的保护措施是需要有所付出的。现在你需要调用显式创建和撤消函数,而且只能通过指针来访问所有的 Date 对象。

程序清单 7.4 一个较安全的 Date 类型(从某种程度上说)

```
/* date2.h */
#ifndef DATE2_H
#define DATE2_H

/* 声明不完全的日期类型 */
```

```
typedef struct Date Date;

Date* date_create(int, int, int);
char* date_format(const Date*, char*);
int date_compare(const Date*, const Date*);
void date_destroy(Date*);

#endif
```

程序清单 7.5 程序清单 7.4 的实现

```
/* date2.c */
#include <stdio.h>
#include <stdlib.h>
#include "date2.h"

struct Date
{
    int month;
    int day;
    int year;
};

static const char *month_text[] =
{ "Bad month", "January", "February", "March", "April",
  "May", "June", "July", "August", "September",
  "October", "November", "December" };

Date* date_create(int m, int d, int y)
{
    Date* dp = malloc(sizeof(Date));
    if (dp == NULL)
        return NULL;

    dp->month = m;
    dp->day = d;
    dp->year = y;
    return dp;
}

char* date_format(const Date* dp, char* buf)
{
    sprintf(buf, "%s %d, %d",
            month_text[dp->month], dp->day, dp->year);
    return buf;
}
```



```

int date_compare(const Date* dp1, const Date* dp2)
{
    int result = dp1->year - dp2->year;
    if (result == 0)
        result = dp1->month - dp2->month;
    if (result == 0)
        result = dp1->day - dp2->day;
    return result;
}

void date_destroy(Date* dp)
{
    free(dp);
}

```

C++至少在三个方面改进了对用户自定义数据类型的支持：（1）它允许在结构体的作用域内定义接口函数；（2）它有直接语言支持禁止用户访问实现的细节；（3）它的构造函数和析构函数能自动执行初始化和清理任务。程序清单 7.7 和 7.8 中的 `Date` 类与程序清单 7.1 和 7.2 中的 `Date` 类有以下几个方面的不同：

1. 数据成员是私有的，所以在用户程序里不能直接地访问它们（只有成员函数的实现才可以访问私有成员）。

2. 接口函数是公有成员函数，因此只能将它们用于 `Date` 对象，并且函数的命名不能损害全局名字空间。因为操作是属于这个类的，所以前缀 `date_` 不再是必要的。注意 `format` 和 `compare` 都是 `const` 成员函数，因为它们不改变 `Date` 的数据成员。

3. 构造函数替代了 C 中结构体的初始化语法，当一个关联对象超出作用域时就会自动调用析构函数来清除对象。

4. 月份名称的文本现在表达在 `struct Date` 的作用域内（是静态成员）。

程序清单 7.9 中的示例程序说明结构体成员使用通常的点运算符来调用成员函数。

程序清单 7.6 测试新的 `Date` 类型

```

/* tdate2.c */
#include <stdio.h>
#include "date2.h"

#define DATELEN 19

main()
{
    Date *d1 = date_create(10,1,1951),
        *d2 = date_create(3,7,1995);
    char buf[DATELEN+1];
    int cmp;

```

```

    printf("d1 == %s\n", date_format(d1, buf));
    printf("d2 == %s\n", date_format(d2, buf));
    cmp = date_compare(d1, d2);
    printf("d1 %s d2\n", cmp < 0 ? "precedes"
                                   : (cmp > 0) ? "follows"
                                   : "equals");

    date_destroy(d1);
    date_destroy(d2);
    return 0;
}

/* 输出:
d1 == October 1, 1951
d2 == March 7, 1995
d1 优于 d2
*/

```

程序清单 7.7 C++ 形式的 Date 类型

```

// date3.h
#ifndef DATE3_H
#define DATE3_H

struct Date
{
    Date(int, int, int);
    char* format(char*) const;
    int compare(const Date&) const;
private:
    int month;
    int day;
    int year;

    static const char* month_text[13];
};

#endif

```

程序清单 7.8 Date 类的实现

```

// date3.cpp
#include <stdio.h>
#include "date3.h"

const char* Date::month_text[13] =
    {"Bad month", "January", "February", "March", "April",

```

```

    "May",        "June",    "July",    "August",  "September",
    "October",    "November", "December"};

Date::Date(int m, int d, int y)
    : month(m),
      day(d),
      year(y)
{}

char* Date::format(char* buf) const
{
    sprintf(buf, "%s %d, %d", month_text[month], day, year);
    return buf;
}

int Date::compare(const Date & dp2) const
{
    int result = year - dp2.year;
    if (result == 0)
        result = month - dp2.month;
    if (result == 0)
        result = day - dp2.day;
    return result;
}

```

程序清单 7.9 举例说明 Date 类

```

// tdate3.cpp
#include <iostream>
#include "date3.h"

main()
{
    const size_t DATELEN = 19;
    char buf[DATELEN+1];

    Date d1(10,1,1951), d2(3,7,1995);
    cout << "d1 == " << d1.format(buf) << endl;
    cout << "d2 == " << d2.format(buf) << endl;
    int cmp = d1.compare(d2);
    cout << "d1 " << (cmp < 0 ? "precedes"
                      : (cmp > 0) ? "follows"
                      : "equals")
          << " d2\n";
}

```

```
// 输出:  
d1 == October 1, 1951  
d2 == March 7, 1995  
d1 precedes d2
```

如果你仍然在为别的用户能看到你的数据成员的构造而烦恼，尽管他并不能访问这些数据成员，可以把数据成员的声明隐藏在一个新的不完全类型中，但是这次对 `Date` 接口是绝对没有影响的（见程序清单 7.10~7.11）。要做到这点，我必须定义 `DateRep` 辅助类，为了高效和简捷，我通过把 `Date` 类变成 `DateRep` 的友元从而让它可以直接访问 `DateRep` 的数据成员。`Date` 构造函数现在需要在堆中创建它的关联 `DateRep` 的对象，因此需要有一个析构函数来释放随后的堆内存。这只需要改变程序清单 7.9 测试程序中的包含语句即可（从 `date3.h` 到 `date4.h`）。用一个定义很好的抽象，就能改变一个实现而不会对用户代码造成很大的影响。

程序清单 7.10 在不完全类型中隐藏 `Date` 的数据

```
// date4.h  
#ifndef DATE4_H  
define DATE4_H  
  
struct DateRep;  
  
struct Date  
{  
    Date(int, int, int);  
    ~Date();  
    char* format(char*) const;  
    int compare(const Date&) const;  
  
private:  
    struct DateRep* drep;  
  
    static const char* month_text[13];  
};  
  
#endif
```

程序清单 7.11 程序清单 7.10 的实现

```
// date4.cpp  
#include <stdio.h>  
#include "date4.h"  
  
struct DateRep  
{  
    DateRep(int, int, int);  
};
```

```

private:
    friend struct Date;

    int month;
    int day;
    int year;
};

const char* Date::month_text[13] =
    {"Bad month", "January", "February", "March", "April",
     "May", "June", "July", "August", "September",
     "October", "November", "December"};
DateRep::DateRep(int m, int d, int y)
    : month(m),
      day(d),
      year(y)
{}

Date::Date(int m, int d, int y)
{
    drep = new DateRep(m,d,y);
}

Date::~~Date()
{
    delete drep;
}

char* Date::format(char* buf) const
{
    sprintf(buf, "%s %d, %d",
            month_text[drep->month],
            drep->day, drep->year);
    return buf;
}

int Date::compare(const Date& dp2) const
{
    int result = drep->year - dp2.drep->year;
    if (result == 0)
        result = drep->month - dp2.drep->month;
    if (result == 0)
        result = drep->day - dp2.drep->day;
    return result;
}

```

7.2 运算符重载

通过给你的类定义中增加运算符函数可以使用户自定义类型像系统预定义类型一样方便地使用。在程序清单 7.12~7.14 中，我已经为简单的输出增加了常用的 6 个关系运算符和一个流插入件。为了提高效率，我同时把较小的函数声明为内联函数（见头文件）。同时也要注意头 `<iosfwd>` 的包含，它为标准流类提供提前声明。由于具有一个 `ostream` 对象引用，因此就没有必要包含 `iostream` 头，它是标准 C++ 库里最大的头文件之一。运算符 `<<` 的实现说明了定义一个流插入器通常仅仅就是将对象成员插入到流中。还要注意我正在用关键字 `class` 来代替 `struct`。它们之间唯一的不同的是 `struct` 的成员缺省时是公有的，而 `class` 的成员缺省时是私有的。

程序清单 7.12 在 `Date` 类中增加运算符

```
// date5.h
#ifndef DATE5_H
#define DATE5_H

#include <iosfwd>
using std::ostream;

class Date
{
public:
    Date(int, int, int);
    int compare(const Date&) const;
    bool operator==(const Date&) const;
    bool operator!=(const Date&) const;
    bool operator<=(const Date&) const;
    bool operator>=(const Date&) const;
    bool operator<(const Date&) const;
    bool operator>(const Date&) const;
    friend ostream& operator<<(ostream&, const Date&);

private:
    int month;
    int day;
    int year;

    static const char* month_text[13];
};

inline bool Date::operator==(const Date& d2) const
{

```

```

    return compare(d2) == 0;
}

inline bool Date::operator!=(const Date& d2) const
{
    return compare(d2) != 0;
}

inline bool Date::operator<=(const Date& d2) const
{
    return compare(d2) <= 0;
}

inline bool Date::operator>=(const Date& d2) const
{
    return compare(d2) >= 0;
} continued
inline bool Date::operator<(const Date& d2) const
{
    return compare(d2) < 0;
}

inline bool Date::operator>(const Date& d2) const
{
    return compare(d2) > 0;
}

#endif

```

程序清单 7.13 增加一个流插入器到 Date 类实现中

```

// date5.cpp
#include <iostream>
#include "date5.h"

const char* Date::month_text[13] =
{ "Bad month", "January", "February", "March", "April",
  "May", "June", "July", "August", "September",
  "October", "November", "December" };

Date::Date(int m, int d, int y)
: month(m),
  day(d),
  year(y)
{}

```

```
ostream& operator<<(ostream& os, const Date& d)
{
    os << Date::month_text[d.month]
        << " " << d.day
        << ", " << d.year;
    return os;
}

int Date::compare(const Date& dp2) const
{
    int result = year - dp2.year;
    if (result == 0)
        result = month - dp2.month;
    if (result == 0)
        result = day - dp2.day;
    return result;
}
```

程序清单 7.14 使用某些 Date 类的比较运算符

```
// tdate5.cpp
#include <iostream>
#include "date5.h"

main()
{
    Date d1(10,1,1951), d2(3,7,1995);

    cout << "d1 == " << d1 << endl;
    cout << "d2 == " << d2 << endl;

    cout << "d1 "
        << (d1 < d2 ? "precedes"
            : (d1 > d2) ? "follows"
            : "equals")
        << " d2" << endl;
}

// 输出:
d1 == October 1, 1951
d2 == March 7, 1995
d1 precedes d2
```

程序清单 7.15 定义一个 Person 数据类型

```
// person.h
```



```

#ifndef PERSON_H
#define PERSON_H

#include <string>
#include "date5.h"

using namespace std;    //字符串

class ostream;

class Person
{
public:
    Person(const string& = "", const string& = "",
           const Date& = Date(0,0,0), const string& = "");
    bool operator==(const Person&) const;
    friend ostream & operator<<(ostream&, const Person&);

private:
    string last;
    string first;
    Date birth;
    string ssn;
};

#endif

```

程序清单 7.16 Person 类的实现

```

// person.cpp
#include <iostream>
#include "person.h"

Person::Person(const string& l, const string& f,
               const Date& b, const string& s)
    : last(l),
      first(f),
      birth(b),
      ssn(s)
{}

ostream& operator<<(ostream& os, const Person& p)
{
    os << '{'
        << p.last << ', '
        << p.first << ', '

```

```

        << '[' << p.birth << ']' << ', '
        << p.ssn
        << '>';
    return os;
}

bool Person::operator==(const Person& p) const
{
    return last == p.last &&
        first == p.first &&
        birth == p.birth &&
        ssn == p.ssn;
}

```

正如在程序清单 7.15 中 `Person` 类的定义, 你可以用其他用户自定义类型或系统预定义类型组成一个新的类型。每一个 `Person` 都有出生日期, 它是一个整个包含在 `Person` 对象中的 `Date` 对象。`Person` 的构造函数通过初始化列表把 `Date` 信息传递到它的子对象中, 该子对象在实现文件 (程序清单 7.16) 中函数头冒号的后面。注意重载运算符 `<<` 是如何隐式地使用 `Date::operator<<` 的。`Person` 类也使用 C++ 标准字符串类作为它的文本数据成员。

7.3 具体的数据类型

由于 C++ 对抽象的支持特别是对运算符重载的支持, 你可以像使用内置类型一样方便地使用自己的数据类型。你对内置类型都能进行哪些典型的处理? 你可以对它们进行初始化、把它们赋值给其他兼容类型的对象, 并且把它们传递给函数或从函数中接收它们。只要你或者编译器提供确定的特殊成员函数, 就可以在 C++ 中用自定义的类型来实现这些行为。这些成员函数的存在组成了所谓的具体的数据类型, 其行为很像内置类型。这些函数包括:

拷贝构造函数

缺省构造函数 (以及其他需要的构造函数)

赋值运算符

析构函数

无论何时创建一个对象都会调用构造函数来对对象进行初始化。初始化参数与构造函数的参数必须在参数个数和类型上相匹配。缺省构造函数是一个不带参数的构造函数。另一个特殊的构造函数叫做拷贝构造函数, 它用一个已经存在的相同类型的对象初始化一个新对象, 而对于某种类型 `T`, 有标志 `T (const T&)` 或 `T (T&)`。无论何时, 当你按值传递或返回一个对象, 或者从已有的对象显式地初始化一个新的对象时, 这个构造函数都会执行。就像下面例子中的 `y`:

```

T x;
...

```

```
T y = x;    // 等同于 "T y(x)"
```

无论何时当用户要把一个对象的值赋给其他已有的对象时，赋值运算符都会执行。它的标志是：

```
T& operator=(const T&)
```

而且它必须总是一个成员函数（而不是一个全局函数）。也可以从其他类型来定义赋值语句，例如从 C 风格字符串转换到日期类型：

```
Date& operator=(const char *);    // 一个可能的日期成员函数
```

注意在程序清单 7.17 第 18 行的赋值语句 (`p3 = p2;`)。我没有显式地定义 `Person::operator=`，但似乎它已经工作了。这是因为编译器已经为我生成它了。在类定义中缺少 `=` 运算符会导致编译器产生一个以成员为单位的赋值，如：

```
Person& Person::operator=(const Person& p)
{
    last = p.last;
    first = p.first;
    birth = p.birth;
    ssn = p.ssn;
    return *this;
}
```

程序清单 7.17 举例说明 Person 类

```
// tperson.cpp
#include <iostream>
#include "person.h" //包括 date5.h
main()
{
    Date d1(12,16,1947);
    Person p1("Richardson", "Alice", d1, "123-45-6789");
    Person p2("Doe", "John");

    cout << "p1 == " << p1 << endl;
    cout << "p2 == " << p2 << endl;
    cout << "p1 " << (p1 == p2 ? "does"
                    : "does not")
        << " equal p2" << endl;

    Person p3;
    p3 = p2;    //18 行
    cout << "p3 " << (p3 == p2 ? "does"
                    : "does not")
        << " equal p2" << endl;
}

// 输出:
```

```

p1 == {Richardson,Alice,[December 16, 1947],123-45-6789}
p2 == {Doe,John,[Bad month 0, 0],}
p1 does not equal p2
p3 does equal p2

```

类似地，如果不提供拷贝构造函数，编译器就会产生如下的拷贝构造函数：

```

Person::Person(const Person& p)
    : last(p.last),
      first(p.first),
      birth(p.birth),
      ssn(p.ssn)
{}

```

就像所见到的，它为每一个成员对象都调用拷贝构造函数。

除了拷贝构造函数（它要单独考虑）外，如果定义了任何构造函数，编译器将不会产生（缺省）构造函数。编译器生成的缺省构造函数为所有的用户定义成员对象调用相应的（缺省）构造函数。一个编译器生成的析构函数调用与任何用户定义成员对象相关联的析构函数。

编译器为 `Person` 类生成的赋值运算符和拷贝构造函数可以工作，因为标准字符串类为这两类函数提供了它自己的版本，而且因为 `Date` 类的数据成员也已建立了。一般而言，只要一个对象的声明完全包含在对象本身内，就不需要重载编译器生成的成员函数了。

例如，在程序清单 7.18 和 7.19 中的 `Person` 类，通过指向 `char` 型的指针把它的文本数据存储在堆中。当真正需要做的事情是给堆分配空间来把文本复制到接收对象时，编译器产生的赋值运算符或拷贝构造函数仅把指针复制给接受对象。当我在这个类里执行测试程序时，Metaware 的 C++ 编译器给出了下面的输出：

```

p1 == {Richardson,Alice,[December 16, 1947],123-45-6789}
p2 == {Doe,John,[Bad month 0, 0],}
p1 does not equal p2
p3 does equal p2
***free(0x4c105c): Pointer already free
Aborting...

```

程序清单 7.18 一个在堆中存储文本有缺陷的 `Person` 类

```

// person2.h
#ifndef PERSON2_H
#define PERSON2_H

#include "date5.h"

class ostream;

class Person
{
public:

```

```

    Person();
    Person(const char*, const char*, const Date&, const char*);
    ~Person();
    bool operator==(const Person&) const;
    friend ostream & operator<<(ostream&, const Person&);

private:
    char* last;
    char* first;
    Date birth;
    char* ssn;
};

#endif

```

在程序出口，编译器调用 `Person` 类的析构函数来销毁 `p3`，并释放堆中的文本内存。由于 `p2` 与 `p3` 所指向的内存相同，当 `p2` 被销毁时析构函数试图再一次删除它。程序清单 7.20 和 7.21 通过增加缺少的成员函数来修复这个错误。

程序清单 7.19 程序清单 7.18 的实现，它缺少赋值运算符和拷贝构造函数

```

// person2.cpp
#include <iostream.h>
#include <string.h>
#include <new.h>
#include "person2.h"

static char* clone(const char* s)
{
    // 返回一个在堆中的字符串拷贝
    char* p = new char[strlen(s) + 1];
    return strcpy(p, s);
}

Person::Person()
    : birth(Date(0,0,0))
{
    last = clone("");
    first = clone("");
    ssn = clone("");
}

Person::Person(const char* l, const char* f,
               const Date& b, const char* s)
    : birth(b)
{
    last = clone(l);
    first = clone(f);
}

```

```
        ssn = clone(s);
    }

    Person::~~Person()
    {
        delete [] last;
        delete [] first;
        delete [] ssn;
    }

    ostream& operator<<(ostream& os, const Person& p)
    {
        os << '{'
            << p.last << ','
            << p.first << ','
            << '[' << p.birth << ']' << ','
            << p.ssn
            << '}'
            << '\n';
        return os;
    }

    bool Person::operator==(const Person& p) const
    {
        return strcmp(last,p.last) == 0 &&
            strcmp(first,p.first) == 0 &&
            birth == p.birth &&
            strcmp(ssn,p.ssn) == 0;
    }
}
```

程序清单 7.20 程序清单 7.18 的运行良好的版本

```
// person3.h
#ifndef PERSON3_H
#define PERSON3_H

#include "date5.h"

class ostream;

class Person
{
public:
    Person();
    Person(const char* ,const char* ,const Date& ,const char*);
    Person(const Person&);           //新的
    ~Person();
};
```

```

        Person& operator=(const Person&); // 新的
        bool operator==(const Person&) const;
        bool operator<(const Person&) const;
        friend ostream& operator<<(ostream&, const Person&);

private:
    char* last;
    char* first;
    Date birth;
    char* ssn;
};

#endif

```

程序清单 7.21 在 Person 类中增加赋值运算符和拷贝构造函数(与程序清单 7.19 比较)

```

//person3.cpp
#include <iostream>
#include <string>
#include <new>           //为了放置新的
#include "person3.h"

static char* clone(const char*);

Person::Person(const Person& p)
    : birth(p.birth)
{
    last = clone(p.last);
    first = clone(p.first);
    ssn = clone(p.ssn);
}

Person& Person::operator=(const Person& p)
{
    if (this != &p)
    {
        last = clone(p.last);
        first = clone(p.first);
        ssn = clone(p.ssn);
    }
    return *this;
}

//其余的与程序清单 7.19 相同

```

7.4 类型抽象

独立于数据类型之外的抽象通常是很有用的。容器就是一个例子。容器是支持其他对象的对象。例如，集合（set）就是一个容器，它可以为成员提供插入、移动和测试元素的操作。程序清单 7.22~7.24 举例说明了实现一个整数集合类型的类。它使用一个固定大小的数组作为内部数据结构（尽管位向量能更有效地实现有序集）。如果想要定义 `SetOfLong` 或者 `SetOfString` 或者用于其他任何支持相等运算符的类型定义集合（set），就会发现唯一改变的东西就是包含在集合中的对象的类型。如果考虑到这一点，一个集合（set）其实不应该非要考虑它支持的对象的类型。

C++ 的模板机制允许用户通过把类型规范为参数来定义一个基本上忽略所包含的对象类型的通用 set 类型。用这样的 set 模板，可以像下面这样请求一系列的整数和字符串集合：

```
set<int> s1;
set<string> s2;
```

第 8 章将介绍有关模板的更多细节。

清单 7.22 使用固定大小数组数据结构的整数集合类型

```
// set.h
#ifndef SET_H
#define SET_H

#include <stddef.h> //为了 size_t

using namespace std

class SetOfInt
{
public:
    SetOfInt();
    bool contains(int) const;
    void insert(int);
    void remove(int);
    void print(ostream&) const;
    friend ostream& operator<<(ostream& os, const SetOfInt& s).
    {
        s.print(os);
        return os;
    }

private:
    enum {LIMIT = 64};
    int elems[LIMIT];
    size_t nelems;
```



```
};
```

```
#endif
```

程序清单 7.23 SetOfInt 的实现

```
// set.cpp
#include <iostream>
#include <algorithm>    //为了 std::find, std::remove (参见第 17 章)
#include "set.h"

using namespace std;

SetOfInt::SetOfInt()
{
    nelems = 0;
}

bool SetOfInt::contains(int x) const
{
    const int* eof = elems + nelems;
    return std::find(elems, eof, x) != eof;
}

void SetOfInt::insert(int x)
{
    if (nelems < LIMIT && !contains(x))
        elems[nelems++] = x;
}

void SetOfInt::remove(int x)
{
    int* eof = elems + nelems;
    if (std::remove(elems, eof, x) != eof)
        --nelems;
}

void SetOfInt::print(ostream& os) const
{
    os << '{';
    for (int i = 0; i < nelems; ++i)
    {
        if (i > 0)
            os << ',';
        os << elems[i];
    }
}
```

```
    os << '}' ;  
}
```

程序清单 7.24 测试 SetOfInt 类

```
// tset.cpp  
#include <iostream>  
#include "set.h"  
  
main()  
{  
    SetOfInt s;  
  
    s.insert(77);  
    s.insert(33);  
    s.insert(500);  
    cout << s << endl;  
  
    s.remove(77);  
    cout << s << endl;  
    cout << "s "  
        << (s.contains(77) ? "does" : "does not")  
        << " contain 77" << endl;  
}  
  
// 输出:  
{77,33,500}  
{33,500}  
s does not contain 77
```

7.5 函数抽象

大概面向对象编程范例的最大的贡献是函数抽象，或者说是众所周知的多态。当提供给用户一个函数接口时，在用户不做任何干涉的情况下函数接口的行为若能根据和它相互作用对象的类型而自动改变，就会产生多态。尽管没有人注意，这实际上发生在机器层。例如，在下面代码中加运算符调用处理器微码中不同的基本算法。

```
int i = 2, j = 3;  
double x = 4.0, y = 5.0;  
int k = i + j;      // 整数加  
int z = x + y;      // 浮点数加
```

同样，如果 f 是一个虚函数，表达式 $p \rightarrow f$ 能调用任何数量的在一个继承层次结构式中相互关联的函数，这取决于基类指针 p 在执行时实际上所指向的对象类型。关于多态的更多内容请参见第 14 章。

7.6 小结

- 复杂性是生活中不可避免的事实。
- 抽象是处理复杂性的工具。
- 定义明确的抽象数据类型可把接口从实现中分离出来。
- C++类支持数据抽象。
- 成员访问控制限定词（`private` 和 `protected`）支持封装。
- 具体的类型应该有定义明确的构造函数、拷贝构造函数、赋值运算符和一个析构函数。
- 运算符重载可以使具体的类型的行为类似于内置类型。
- 模板支持类型抽象。
- 虚函数支持函数抽象。

模 板

第1章介绍了以引用作为函数参数来交换整数：

```
#include <iostream>
using namespace std;

void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}

main()
{
    int a = 1, b = 2;
    swap(a,b);
    cout << "a == " << a << ", b == " << b << endl;
}

//输出：
a == 2, b == 1
```

如果需要交换其他类型的对象该怎么办？当然可以利用函数重载，并且可以为可能需要的每个数据类型都提供一个不同版本的交换函数 `swap()`，如下：

```
void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}
```

```
}

void swap(float& x, float& y)
{
    float temp = x;
    x = y;
    y = temp;
}

void swap(char*& x, char*& y)
{
    char* temp = x;
    x = y;
    y = temp;
}
```

现在可以用相同的方法为每一个不同的交换对调用交换函数 (swap):

```
#include <iostream>
using namespace std;

main()
{
    int a = 1, b = 2;
    float c = 100.0, d = 200.0;
    char *s = "hello", *t = "goodbye";

    swap(a,b);
    cout << "a == " << a << ", b == " << b << endl;
    swap(c,d);
    cout << "c == " << c << ", d == " << d << endl;
    swap(s,t);
    cout << "s == " << s << ", t == " << t << endl;
}

// 输出:
a == 2, b == 1
c == 200, d == 100
s == goodbye, t == hello
```

立即就得到两条观察结果:

1. 当需要处理一个新的类型时, 需要对一个交换函数 swap() 建立新的重载并且重建软件。

2. 每个重载的逻辑是相同的, 只是数据类型不同。

如果能仅用一次交换函数 swap() 的过程, 就可以准确地替换不同类型的参数, 这难道

不好吗？

8.1 泛型编程

在这里我们需要的是支持类型抽象，它是一个允许我们将数据类型作为参数使用的机制。对于你，一个以“有困难时使用预处理器”为哲学思想的不计后果的开发者，你可能会想到把类型变成一个宏参数。下面的宏会为任意版本的交换函数 `swap()` 产生代码。

```
// swap1.h:
#define genswap(T)      \
void swap(T& x, T& y)   \
{                       \
    T temp = x;         \
    x = y;              \
    y = temp;           \
}
```

现在测试程序成为：

```
#include <iostream>
#include "swap1.h"
using namespace std;

// 生成所需的代码：
genswap(int)
genswap(float)
genswap(char*)

main()
{
    // (与前面相同)
}
```

尽管这个程序可以运行，但是那些不用分号结尾的宏调用语句无疑看起来是奇怪的。而且仍有忘记实例化自己所需要的所有重载的可能性，而需要回头重复从编辑到编译程序的过程。此外，难道类函数的宏不能假设为在 C++ 中消失吗？为什么不能让编译器来为你做所有的工作呢？

8.2 函数模板

当然你能让编译器来做所有的工作。函数模板是一个形式语言机制，这个机制为一个从实际函数调用的上下文中引用适当的数据类型的函数自动地生成代码。交换函数 `swap()` 的模板如下：

```
// swap2.h (模板版本)
template<class T>
void swap(T& x, T& y)
{
    T t = x;
    x = y;
    y = t;
}
```

前缀 `template<class T>` 告诉编译器而后的函数定义把 `T` 当作类型参数使用。当编译器遇到一个实际的交换函数 `swap()` 的调用时, 例如:

```
swap(a,b);
```

编译器从实参 `a,b` 的类型推断出 `T`, 然后为:

```
void swap(int& x,int& y);
```

产生代码, 现在可以将引用移到 `genswap` 宏处, 然后运行如下的测试程序:

```
#include <iostream>
#include "swap2.h"
```

```
using std::cout;
using std::endl;
main()
{
    // (与前面相同)
}
```

因为交换函数 `swap()` 模板在标准库中已经被定义了, 因此, 我不得不将语句:

```
using namespace std;
```

替换为如下语句:

```
using std::cout;
using std::endl;
```

以避免 `std::swap` 和我的交换函数 `swap()` 模板发生冲突。想要了解更多关于名字空间和 `using` 关键字的信息, 请参见第 11 章。当然也可以重新命名交换函数 `swap()`。

注意交换函数 `swap()` 模板的实体已被全部包含在一个单独的包含文件中了。通常不在那里定义非内联函数, 但记住交换函数 `swap()` 并非一个函数而是一个模板, 好像是一个编译器用来按要求生成函数的模式。在一个专门的包含文件中包含所有的模板代码是习惯的做法。(当这本书出版时, 这个模板编译的包含模式是在微软 Windows 平台中支持的, 而大多数的 UNIX 编译器支持分离模式, 在那里函数实体可以存在于一个分离的实现文件中。我在本书中只用包含模式)。

要定义一个具有参数在函数参数列表中但不使用的函数模板, 这是能够做到的。

例如,

```
template<class To, class From>
```



```

To convert(const From& f)
{
    return To(f);
}

```

由于函数是根据函数调用中的实参来区分的，因此在 `convert` 的调用中没有办法推断参数 `To` 的类型。在这种情况下，显式地限定函数调用来说明需要哪个实例。

```

string s = convert<string> ("foo")
string s = convert<string, char*> ("bar");    //char*可选择

```

`convert` 的第一次调用从实参“foo”推断出第二个模板参数 `From` 为 `char *`型。就像第二次调用所说明的那样，你还能显式地限定这两个类型。

8.3 类模板

C++也允许类模板（class templates），其中类使用的类型可以作为模板参数出现。例如，在第7章的 `SetOfInt` 类中，操作不依赖它所支持的对象类型。可以通过用一个模板参数（见程序清单 8.1）替代所有的整型实例来定义一个 `Set` 模板（当它作为 `SetOfInt` 里的包含对象类型使用时）。为了实例化模板类，要在括号内指定所需要的类型，如下：

```

Set<int> s1;
Set<float> s2;
Set<string> s3;

```

一个类模板中定义的成员函数成为有效的函数模板，如程序清单 8.1 所示。注意在任何需要类名的地方出现的符号 `Set<T>`。这里因为 `Set` 是一个模板，不是一个类（`Set<T>`代表一个类）。程序清单 8.2 和程序清单 8.3 的测试程序分别地使用了整型和字符串类型集合（`Set<int>`和 `Set<string>`）。

程序清单 8.1 一个 `Set` 类模板

```

// set.h
#ifndef SET_H
#define SET_H
#include <iostream>
#include <algorithm>
#include <stddef.h>

template<class T>
class Set
{
public:
    Set();
    bool contains(const T&) const;
    void insert(const T&);
    void remove(const T&);

```

```
    void print(std::ostream&) const;
private:
    enum {LIMIT = 64};
    T elems[LIMIT];
    size_t nelems;
};

template<class T>
Set<T>::Set()
{
    nelems = 0;
}

template<class T>
bool Set<T>::contains(const T& x) const
{
    const T* eof = elems + nelems;
    return std::find(elems, eof, x) != eof;
}

template<class T>
void Set<T>::insert(const T& x)
{
    if (nelems < LIMIT && !contains(x))
        elems[nelems++] = x;
}

template<class T>
void Set<T>::remove(const T& x)
{
    T* eof = elems + nelems;
    if (std::remove(elems, eof, x) != eof)
        --nelems;
}

template<class T>
void Set<T>::print(std::ostream& os) const
{
    os << '{';
    for (int i = 0; i < nelems; ++i)
    {
        if (i > 0)
            os << ',';
        os << elems[i];
    }
    os << '}';
}
```

```
// 全局函数:
template<class T>
std::ostream& operator<<(std::ostream& os, const Set<T>& s)
{
    s.print(os);
    return os;
}

#endif
```

程序清单 8.2 测试一组整数

```
// tset.cpp
#include <iostream>
#include "set.h"
using namespace std;

main()
{
    Set<int> s;

    s.insert(77);
    s.insert(33);
    s.insert(500);
    cout << s << endl;
    s.remove(77);
    cout << s << endl;
    cout << "s "
         << (s.contains(77) ? "does" : "does not")
         << " contain 77" << endl;
}

//输出:
{77,33,500}
{33,500}
s does not contain 77
```

程序清单 8.3 测试一组字符串

```
// tsets.cpp
#include <iostream>
#include <string>
#include "set.h"
using namespace std;

main()
```

```

{
    Set<string> s;

    s.insert("one");
    s.insert("two");
    s.insert("three");
    cout << s << endl;

    s.remove("three");
    cout << s << endl;
    cout << "s "
        << (s.contains("three") ? "does" : "does not")
        << " contain \"three\"" << endl;
}

```

//输出:

{one,two,three}

{one,two}

s does not contain "three"

偶尔, 用户可能需要定义一个其本身是一个独立模板的成员函数。这个成员模板工具允许这样做:

```

#include <iostream>
using namespace std;

template<class T>
class A
{
public:
    A()
    {
        cout << "default ctor\n";
    }
    template<class B> A(const B& b)
    {
        cout << "converting from " << b << "\n";
    }
    template<class C> void f(C c)
    {
        cout << c << endl;
    }
};

```

在下面的测试程序中, 声明 `a` 时调用默认构造函数 `A<int>::A()`。调用 `a.f("hello")` 实例化 `void A<int>::f(char*)`, 接着调用实例化 `void A<int>::f(int)`。定义 `a2` 要求构造函数

`A<float>::A (const int &)`，而最后一个函数调用为 `void A<float>::f (char)` 创建代码。模板参数 `C` 的实参必须和 `std::ostream::operator<<` 兼容，否则编译器将会发出诊断。

```
main()
{
    A<int> a;
    a.f("hello");
    a.f(1);

    A<float> a2(120);
    a2.f('c');
}

//输出:
default ctor
hello
1
converting from 120
c
```

为了方便，我在 `situ` 中定义上面 `A` 的成员，这意味着函数体出现在自己的类模板定义中。为了在 `A` 定义的外部定义 `f`，需要用下面语法：

```
template<class T> template<class C>
void A<T>::f(C c) {...}
```

8.4 模板参数

除了数据成员和成员函数外，类也能包含被定义为嵌套类或 `typedefs` 的类型。考虑下面的类声明：

```
#include <iostream>
using namespace std;

class Foo
{
public:
    typedef int U;
};

class Bar
{
public:
    typedef char* U;
};

template<class T>
```

```

class Baz
{
    typename T::U x;
public:
    Baz(const typename T::U& t) : x(t){};
    void f() {cout << x << endl;}
};

main()
{
    Baz<Foo> b1(1);
    b1.f();

    Baz<Bar> b2("hello!");
    b2.f();
}

// 输出:
1
hello!

```

编译器很容易对诸如 `Baz<T>::X` 之类的声明产生混淆。无论何时当采用类似 `T::U` 的用法时，要使用关键字 `typename` 告诉编译器 `T::U` 是一个类型，其中 `T` 是一个模板类型参数，而 `U` 是一个类型。你也可以在形式模板参数声明中使用 `type name` 代替 `class`。

模板也可以有无类型参数，其中最常见的是作为容器维数使用的整数值。程序清单 8.4 定义了一个固定大小的 `Set` 模板，其中第二个模板参数 (`LIMIT`) 成为下面数组的维数。可以声明如下的 `Set`:

```
Set<int, 10> s;
```

无类型模板实参必须是编译期常数表达式，这个表达式的值可以是一个整型值（包括枚举常量）也可以是一个指针，或者是一个引用，但不允许是浮点值。

程序清单 8.4 一个带有无类型模板参数的 `Set` 模板

```

// set2.h

#ifndef SET_H
#define SET_H
#include <iostream>
#include <algorithm>
#include <stddef.h>

template<typename T, size_t LIMIT>
class Set
{
public:

```

```

    Set();
    bool contains(const T&) const;
    void insert(const T&);
    void remove(const T&);
    void print(std::ostream&) const;

private:
    T elems[LIMIT];
    size_t nelems;
};

template<typename T, size_t LIMIT>
Set<T, LIMIT>::Set()
{
    nelems = 0;
}

// 等等。

```

也可以使用缺省模板参数，这样用户就可以选择或指定容器的大小，或者就用的值。在这种情况下，应该按如下方式定义 Set：

```

template<typename T, size_t LIMIT = 10>
class Set
{
    // 等等。

```

并且声明

```
Set<int> s;
```

将会把 LIMIT 设定为 10。

缺省模板参数也可以是类型参数。例如在 C++ 标准库中，实际的设置模板声明是这样开始的：

```

template<class Key, class Compare = less<Key>,
        class Allocator = allocator<Key> >
class set {...

```

除非用户指定了 compare 函数对象和一个分配器，否则将会实例化默认模板参数（要了解更多关于标准容器的内容，请参见第 16 章）。

8.5 模板特化

可以想象得到，你可能偶然想重写由一个模板产生的代码。例如，考虑下面用于比较两个对象的函数模板：

```

template<class T>
int comp(const T& t1, const T& t2)

```

```
{  
    return (t1 < t2) ? -1 : (t1 == t2) ? 0 : 1;  
}
```

当然，用 `char*` 型（或者出于这个目的，使用任何类型的指针）实例化 `comp` 将不会正确进行。为了让 `comp` 具备 C 风格字符串的特点，可以提供显式特化，如下所示：

```
template<>  
int comp<const char*>(const char*& t1, const char*& t2)  
{  
    return strcmp(t1,t2);  
}
```

只要这个特化在调用具有 C 风格字符串参数的 `comp` 之前声明了，它就将重写来自原始模板的缺省实例。程序清单 8.5 中的程序表明可以用多种语法不同的方式来特化模板，包括简单的函数重载，但是为了风格的统一推荐使用上面的显式特化，也可以特化类（见程序清单 8.6）。

程序清单 8.5 函数模板特化举例

```
// spec.cpp  
#include <iostream>  
#include <string.h>          //标准 C  
#include <string>  
  
using namespace std;  
  
template<class T>  
size_t bytes(T& t)  
{  
    cout << "(using primary template)\n";  
    return sizeof t;  
}  
  
size_t bytes(char*& s)  
{  
    cout << "(using char* overload)\n";  
    return strlen(s) + 1;  
}  
  
size_t bytes<wchar_t*>(wchar_t*& w)  
{  
    cout << "(using wchar_t* specialization)\n";  
    return 2*(wcslen(w) + 1);  
}  
  
template<>
```



```

size_t bytes<>(string& s)
{
    cout << "(using string explicit specialization)\n";
    return sizeof s;
}

template<>
size_t bytes<float>(float& x)
{
    cout << "(using float explicit specialization)\n";
    return sizeof x;
}

main()
{
    int i;
    cout << "bytes in i: " << bytes(i) << endl;
    char *s = "hello";
    cout << "bytes in s: " << bytes(s) << endl;
    wchar_t *w = L"goodbye";
    cout << "bytes in w: " << bytes(w) << endl;
    string t;
    cout << "bytes in t: " << bytes(t) << endl;
    float x;
    cout << "bytes in x: " << bytes(x) << endl;
    double y;
    cout << "bytes in y: " << bytes(y) << endl;
    return 0;
}

//输出:
(using primary template)
bytes in i: 4
(using char* overload)
bytes in s: 6
(using wchar_t* specialization)
bytes in w: 16
(using string explicit specialization)
bytes in t: 16
(using float explicit specialization)
bytes in x: 4
(using primary template)
bytes in y: 8

```

程序清单 8.6 类模板特化举例

```
// spec2.cpp
```

```
#include <iostream>
using namespace std;

template<typename T>
class A
{
public:
    A() {cout << "primary\n";}
};

class A<char>
{
public:
    A() {cout << "char specialization\n";}
};

template<>
class A<float>
{
public:
    A() {cout << "float specialization\n";}
};

main()
{
    A<int> a1;
    A<char> a2;
    A<float> a3;
}

//输出:
primary
char specialization
float specialization
```

只特化部分类模板参数也是可能的。程序清单 8.7 中，只要第一个模板参数是指向任意类型的指针，局部特化 $A<T^*, U>$ 就将用于实例化对象 A ，而且只要参数类型是相同的， $A<T, T>$ 将被调用。无论何时只要看到类型的角括号后紧跟类模板名，就要知道这里是模板特化。正如上面所看到的那样，一些完全特化并不需要它们，但是局部特化却总是需要它们的。C++ 中不支持函数模板的部分特化。

程序清单 8.7 类模板的部分特化举例

```
// partial.cpp
#include <iostream>
using namespace std;
```

```

template<class T, class U>
class A
{
public:
    A() {cout << "primary template\n";}
};

template<class T, class U>
class A<T*, U>
{
public:
    A() {cout << "<T*,U> partial specialization\n";}
};

template<class T>
class A<T, T>
{
public:
    A() {cout << "<T,T> partial specialization\n";}
};

template<class T, U>
class A<int, U>
{
public:
    A() {cout << "<int, U> partial specialization\n";}
};

main()
{
    A<char, int> a1;
    A<char*, int> a2;
    A<float, float> a3;
    A<int, float> a4;
}

//输出:
primary template1
<T*,U> specialization
<T,T> specialization
<int,U> specialization

```

¹ 译注: <int,U> specialization // 此程序在 VC++6.0 中, 调试错误, A<int,U>, A<T*, U>, A<int, U>: template class has already been defined as a non-template class,unrecognizable template declaration/definition,VC 不支持此种方式的定义。

8.6 小结

- 模板实现类型抽象，可以将模板工具看作是将类型作为参数的编译期机制。
- 当在代码中遇到相关函数调用时，编译器会从函数模板中实例化一个函数，从函数调用的实参中推断它的类型参数（除非显式地提供了它们）。
 - 类是当用显式类型参数声明它们时从类模板中实例化的。
 - 在模板声明中既能提供类型参数，又能提供非类型参数。非类型参数必须是编译期的常量表达式。
 - 允许使用缺省模板参数。就像缺省函数参数一样，缺省模板参数只能作为尾部参数出现在模板声明中。
 - 可以通过模板特化来为模板参数的具体设置重写模板实例化机制。只有一些模板参数被特化的部分特化也是允许的，并且在模板名字后面角括号的代码也能被识别出来。

位操作

在系统编程中，操作一个整数的各个位的能力是很有必要的。例如在 MSDOS 和 UNIX 文件系统中，每个磁盘文件都有相关的属性字节。可以保护删除一个文件或者可以把它从目录清单中永久删除，这些都依赖于设置的属性字节。在 C 时代开始前，大多数的程序员不得不为了实现位操作而求助于汇编语言。现在，随着逐位运算符和大多数编译器提供的主机系统接口的出现，在一个 C 程序里几乎没有什么事情是操作系统办不到的。在本章里，我将复习一下常见的位操作技术，然后介绍一下标准 C++ 库中 `bitset<N>` 和 `vector<bool>` 类模板。

9.1 按位运算符

共有 6 种按位运算符（见表 9.1）。如程序清单 9.1 所示，如果操作数中相应的两个位中任何一个为 1，则按位“或”运算符将结果置为 1。例如，

```
01010101    (== 0x55)
| 11110000    (== 0xf0)
-----
= 11110101    (== 0xf5)
```

按位“与”操作运算符仅当操作数中相应的位都是 1 时才置 1。

```
01010101
& 11110000
-----
= 01010000    (== 0x50)
```

程序清单 9.1 举例说明按位运算符

```
// bit1.cpp
#include <iostream>
```

```

#include <iomanip>
using namespace std;

main()
{
    typedef unsigned int word;
    typedef unsigned char byte;
    byte a = 0x55;
    byte b = 0xf0;

    cout.setf(ios::hex, ios::basefield);
    cout.setf(ios::uppercase);
    cout.fill('0');

    cout << "a | b == " << setw(2) << word(a | b) << endl;
    cout << "a & b == " << setw(2) << word(a & b) << endl;
    cout << "a ^ b == " << setw(2) << word(a ^ b) << endl;
    cout << "~a == " << setw(2) << word(byte(~a)) << endl;
    cout << "a << 1 == " << setw(2) << word(a << 1) << endl;
    cout << "b >> 6 == " << setw(2) << word(a >> 6) << endl;
}

//输出:
a | b == F5
a & b == 50
a ^ b == A5
~a == AA
a << 1 == AA
b >> 6 == 01

```

表 9.1

按位运算符

操 作 符	意 义
&	按位与
	按位或
^	按位异或
~	按位求补码
<<	左移
>>	右移

按位运算符“异或”的作用是当原始操作数中相应的两个位不不同时（即这两个位的值一个为 1，另一个为 0 时），将该位置 1。

```

    01010101
^   11110000
-----
=   10100000    (== 0xa0)

```

按位“非”运算符将每个位取反，即 1 变成 0，反之也是如此。程序清单 9.1 中成整型 (int) 的强制类型转换 (即 byte) 是必要的，因为 cout 会将 char 作为字符而不是数字打印。对无符号字符型的强制类型转换 (即 byte) 是必要的，因为标准—相容编译器总是在应用位操作之前就将字符型变成整型。若没有强制类型转换，则输出为：

```
~a == FFFFFFFA
```

因为标准输出机制从不截断数字中的有效值，所以在我的工作平台上，整型占四个字节。移位运算符按右操作数指定的位数把所有的位一起向左或向右移动。“落在结尾后面”的位被舍去。左移运算符对所有的空位重新设置。当有符号整数右移时，一些机器会补零，而其他机器会复制符号位。因此，在位操作中，习惯做法是把整数声明为无符号的 (unsigned)，当将无符号数右移时，总是补零。

9.2 访问单独的位

你经常需要设置、重设置或检验单独的位。如果不涉及太多的位，可能会考虑使用位字段结构。例如，假设只对一个整数里的第 4 位或第 5 位感兴趣。逻辑上讲，位是从右向左数的，而且从零开始 (即每位代表了这个数二进制表示的每一位上 2 的幂)。在一个从小到大的字节存储顺序中 (关于“结尾顺序”的详细内容，请参见第 2 章)，以下结构体将会很适用：

```

struct bits4_5
{
    unsigned : 4;          /* 从位 0 向上跳过到位 3 */
    unsigned bit4 : 1;
    unsigned bit5 : 1;
};

```

一个从大到小的字节存储顺序结构体要求：

```

struct bits4_5
{
    unsigned int : 10;     /* 从位 15 向下跳过到 6 位 */
    unsigned int bit5 : 1;
    unsigned int bit4 : 1;
};

```

可以通过命名的变量来处理位：

```

unsigned int x;
struct bits4_5 *xp = (struct bits4_5 *) &x;
xp->bit4 = 0;          /* 复位位 4 */

```

```

xp->bit5 = 1;    /* 置位 5 */
f(xp);          /* f 有可能修改位 */
if (xp->bit4)    /* 位 4 仍然被设置了吗? */
    ...

```

然而这项技术有点不太诱人：它缺乏可移植性而且使处理大量位变得很笨重。能任意操作单独位的首选方法就是通过按位操作符。例如要设置一个整数的第 i 位，形成一个屏蔽，使它的第 i 位为 1 其他位均为 0（我把它叫做 1 屏蔽），然后对这个整数进行按位“或”运算：

```
x |= (1u << i);
```

为了对位取反，形成上面屏蔽的补（0 屏蔽），对这个整数进行按位“与”运算：

```
x &= ~(1u << i);
```

程序清单 9.2 中的程序依次在逻辑上从右到左对一个整数的所有位置位，又以同样的顺序对它们进行复位。量 `CHAR_BIT` 是一个字节中位的个数，它来自 `<limits.h>`。

对位进行切换，把 1 屏蔽异或成下面的数：

```
x ^= (1u << i);
```

为了知道一个位是否被置位，可以用相同的屏蔽对整数进行“或”运算：

```
x &= (1u << i);
```

如果该位没有被置位，表达式返回零值，否则返回非零值（实际上是 2 的 i 次幂）。下面有用的习惯用法只返回 0 或 1，以便于将结果作为数组的下标。

```
!!(x & (1u << i));
```

程序清单 9.3 中的头文件为这些操作的非析构版本定义了宏，并且声明一些有用的函数。程序清单 9.4 中的函数实现使用 `nbits` 宏按位计算无符号整数的大小。函数 `fputb` 以二进制形式打印一个无符号整数（unsigned interger），并且右边补零，如果需要左边也补零。函数 `fgetb` 读出一串位字符，然后将它变成无符号数。这个奇怪的语句：

```
sprintf(format, "%d[01]", nb);
```

建立了一个跳过空格的扫描格式字符串，然后累计 1 或 0 到 `nb` 中直到它扫到了一个非位字符为止。用户可能会认为在扫描中通过使用可变宽度的格式描述符可以避免 `sprintf`：

```
fscanf(f, "%*[01]", nb, buf);
```

不幸的是，可变宽度替换不能和扫描集一起工作的。（关于扫描集和相关文本处理更多的内容，请参见第 17 章）。

`count` 函数通过一次右移一位并测试 0 位来计算一个数中 1 的个数。程序清单 9.5 的测试程序说明了这些函数。

清单 9.2 置位和复位单独的位

```

// bit2.cpp: 置位和复位单独的位
#include <iostream>
#include <iomanip>
#include <climits>    // <limits.h>
using namespace std;

```



```
main()
{
    typedef unsigned short word;
    const size_t NBYTES = sizeof(word);
    const size_t NBITS = NBYTES * CHAR_BIT;
    const size_t NXDIGITS = NBYTES * 2;
    word n = 0;

    cout.setf(ios::hex, ios::basefield);
    cout.setf(ios::uppercase);
    cout.fill('0');

    // 依次设置每一位:
    for (int i = 0; i < NBITS; ++i)
    {
        n |= (1u << i);
        cout << setw(NXDIGITS) << n << endl;
    }
    cout << endl;

    // 现在关闭它们:
    for (int i = 0; i < NBITS; ++i)
    {
        n &= ~(1u << i);
        cout << setw(NXDIGITS) << n << endl;
    }
}

//输出:
0001
0003
0007
000F
001F
003F
007F
00FF
01FF
03FF
07FF
0FFF
1FFF
3FFF
7FFF
FFFF
```

FFFE
FFFC
FFF8
FFF0
FEE0
FFC0
FF80
FF00
FE00
FC00
F800
F000
E000
C000
8000
0000

程序清单 9.3 位存取函数声明

```
/* bit.h: 用于无符号整型的位函数 */
#ifndef BIT_H
#define BIT_H

#include <stdio.h>
#include <limits.h>

#define mask1(i)    (1u << i)
#define mask0(i)    ~(1u << i)

#define set(n,i)     ((n) | mask1(i))
#define reset(n,i)   ((n) & mask0(i))
#define toggle(n,i)  ((n) ^ mask1(i))
#define test(n,i)    !((n) & mask1(i))

#define nbits(x) (sizeof(x) * CHAR_BIT)

unsigned fputb(unsigned, FILE *);
unsigned fgetb(FILE *);
unsigned count(unsigned);

#endif
```

程序清单 9.4 bit.h 的实现文件

```
/* bit.c: 用于无符号整型数的位运算符 */
#include <stdio.h>
```

```
#include <limits.h>
#include <stdlib.h>
#include <string.h>
#include "bit.h"

unsigned fputb(unsigned n, FILE *f)
{
    int i;
    size_t nb = nbits(n);

    /* 打印二进制形式的数 */
    for (i = 0; i < nb; ++i)
        fprintf(f, "%d", test(n, nb-1-i));
    return n;
}

unsigned fgetb(FILE *f)
{
    unsigned n = 0;
    size_t nb = nbits(n);
    size_t slen;
    char *buf = malloc(nb+1);
    char format[9];
    int i;

    if (buf == NULL)
        return 0;

    /* 构造读格式(例如, " %16[01]") */
    sprintf(format, " %16[01]", nb);
    if (fscanf(f, format, buf) != 1)
        return 0;
    /* 在 n 中设置相应的位 */
    slen = strlen(buf);
    for (i = 0; i < slen; ++i)
        if (buf[slen-1-i] == '1')
            n = set(n, i);

    free(buf);
    return n;
}

unsigned count(unsigned n)
{
    unsigned sum = 0;
```

```
while (n)
{
    if (n & 1u)
        ++sum;
    n >>= 1;
}
return sum;
}
```

程序清单 9.5 从 bit.h 说明位存取函数

```
/* tbit.c: 使用 bit.h 的位操作 */
#include <stdio.h>
#include "bit.h"
main()
{
    int i;
    unsigned int n = 0;
    size_t nb = nbits(n);

    /* 设置偶数位 */
    for (i = 0; i < nb; i += 2)
        n = set(n,i);
    printf("n == %08X (" ,n);
    fputb(n,stdout);
    printf("), count == %d\n",count(n));

    /* 将高位取反 (上面的一半) */
    for (i = nb/2; i < nb; ++i)
        n = toggle(n,i);
    printf("n == %08X (" ,n);
    fputb(n,stdout);
    printf("), count == %d\n",count(n));

    /* 复位低位 (下面的一半) */
    for (i = 0; i < nb/2; ++i)
        n = reset(n,i);
    printf("n == %08X (" ,n);
    fputb(n,stdout);
    printf("), count == %d\n",count(n));

    /* 读一个位字符串 */
    fputs("Enter a bit string: ",stderr);
    n = fgetb(stdin);
    printf("n == %08X (" ,n);
    fputb(n,stdout);
    printf("), count == %d\n",count(n));
}
```

```

    return 0;
}

/* 示例执行:
n == 55555555 (01010101010101010101010101010101), count == 16
n == AAAA5555 (10101010101010100101010101010101), count == 16
n == AAAA0000 (10101010101010100000000000000000), count == 8
Enter a bit string: 1010010001000010000010000001
n == 0A442081 (00001010010001000010000010000001), count == 7
*/

```

9.3 大型置位

因为标准 C/C++ 保证长整型至少有 32 位，所以无符号长整型通常足以容纳在一个应用程序中需要的所有标志。程序清单 9.6 和程序清单 9.7 说明了 Bitmask 类，它是一个有效且易懂的 C++ 类，使你可以很容易地访问长整型中的位。但是，如果需要的位数超过了一个整数所能容纳的位数将会怎样？例如，假设必须创建一个“选取表”用户接口对象。选取表是滚动的、弹出式的、允许用户选择多种条目的列表框。条目的数目很容易超过长整数中位的个数。跟踪每个条目状态的最有效的方法是将置位与选取表相联系：如果用户重视第 i 个条目，那么可以对第 i 位进行置位。

程序清单 9.6 一个 32 位的 Bitmask 类

```

// bitmask.h: 无符号长整型数中的位存取
#include <stddef.h>
#include <assert.h>
#include <limits.h>

#if !defined(BITMASK_H)
#define BITMASK_H

class Bitmask
{
    typedef unsigned long _Block;

public:
    explicit Bitmask(_Block n = 0ul);
    bool test(size_t pos) const;
    void set(size_t pos);
    void set1(size_t pos, bool val);
    void reset(size_t pos);
    void reset();

```

```
    Bitmask& operator|=(const Bitmask&);
    Bitmask& operator&=(const Bitmask&);
    friend Bitmask operator|(const Bitmask&, const Bitmask&);
    friend Bitmask operator&(const Bitmask&, const Bitmask&);
    Bitmask operator~() const;

    operator _Block() const;
    size_t size() const;

private:
    Block m_Bits;
};

inline Bitmask::Bitmask(_Block n)
{
    m_Bits = n;
}

inline bool Bitmask::test(size_t pos) const
{
    assert(0 <= pos && pos < size());
    return m_Bits & (1ul << pos);
}

inline void Bitmask::set(size_t pos)
{
    assert(0 <= pos && pos < size());
    m_Bits |= (1ul << pos);
}

inline void Bitmask::set1(size_t pos, bool val)
{
    val ? set(pos) : reset(pos);
}

inline void Bitmask::reset(size_t pos)
{
    assert(0 <= pos && pos < size());
    m_Bits &= ~(1ul << pos);
}

inline void Bitmask::reset()
{
    m_Bits = 0ul;
}
```

```
inline Bitmask& Bitmask::operator|=(const Bitmask& b)
{
    m_Bits |= b.m_Bits;
    return *this;
}

inline Bitmask& Bitmask::operator&=(const Bitmask& b)
{
    m_Bits &= b.m_Bits;
    return *this;
}

inline Bitmask operator|(const Bitmask& b1, const Bitmask& b2)
{
    return Bitmask(b1.m_Bits | b2.m_Bits);
}

inline Bitmask operator&(const Bitmask& b1, const Bitmask& b2)
{
    return Bitmask(b1.m_Bits & b2.m_Bits);
}

inline Bitmask Bitmask::operator~() const
{
    return Bitmask(~m_Bits);
}

inline Bitmask::operator Bitmask::_Block() const
{
    return m_Bits;
}

inline size_t Bitmask::size() const
{
    return sizeof(_Block) * CHAR_BIT;
}

#endif
```

程序清单 9.7 测试 Bitmask 类

```
// tbitmask.cpp: 测试 Bitmask 类
#include "bitmask.h"
#include <iostream>
using namespace std;
```

```
main()
{
    Bitmask b;

    cout.setf(ios::boolalpha);
    cout.setf(ios::hex, ios::basefield);

    b.set(1);
    b.set(3);
    cout << b << endl;

    b.reset(1);
    cout << b << endl;
    cout << b.test(1) << endl;
    cout << b.test(2) << endl;
    cout << b.test(3) << endl;

    b |= Bitmask(0xabcdef00);
    cout << b << endl;

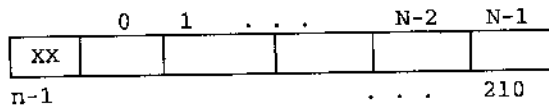
    // 以二进制字符串形式打印全部的 bitmask:
    for (int i = b.size()-1; i >= 0; --i)
        cout << (b.test(i) ? '1' : '0');
    cout << endl;
}
```

```
//输出:
```

```
a
8
false
false
true
abcdef08
10101011110011011110111100001000
```

当然也可以使用无符号整数数组存放所需要的位：

数组索引：



位的位置:

如果需要 n 个位，那么就需要 $N = \text{ceil}(n/\text{blksiz})$ 个数组元素，其中 **BLKSIZ** 是无符号整型（或者是数组里的整型）的位数。如果 n 不是 **BLKSIZ** 的倍数，那么在第 0 块中将会有未使用的位（用上面的 **XX** 代表）。现在处理特殊位就简化成了找到位所在的特殊字块并计算用于该块屏蔽的偏移。为了找到正确的块，注意下面的模式。

在该范围内的位	在该块中	及 b/BLKSIZ==
[0,BLKSIZ-1]	N-1	0
[BLKSIZ,2*BLKSIZ-1]	N-2	1
...
[(N-1)*BLKSIZ,n-1]	0	N-1

如果 b 是正在被讨论的位数，那么，

$B + b/\text{BLKSIZ} == N-1$

这里 B 是所要找的块数。因此，

$B = N-1 - b/\text{BLKSIZ}$

在 B 块内的位 b 的偏移量是：

$\text{offset} = b \% \text{BLKSIZ}$

如果 bits 是数组名，那么可以用这些表达式处理第 b 位：

$\text{bits}[B] |= (\text{lu} \ll \text{offset});$ /* 置位 */

$\text{bits}[B] \&= \sim(\text{lu} \ll \text{offset});$ /* 复位 */

$\text{bits}[B] \wedge= (\text{lu} \ll \text{offset});$ /* 取反 */

$!!\text{bits}[B] \& (\text{lu} \ll \text{offset});$ /* 测试 */

程序清单 9.8 中的头文件表示了处理任意大型置位的接口。除了已经讨论过的函数外，它又增加了转换成无符号数和由无符号数转换而来的功能，以便于你只用字大小置位并且想要和主机环境连接。语句

```
typedef struct bits Bits;
```

把 `struct bits` 声明为和 `Bits` 是同义词的不完全类型。结构体的定义在实现文件中（见程序清单 9.9，关于不完全类型的讨论，见第 2 章）。如果想要在基数组里使用另外的整数类型，只需在 `bits.c`（见程序清单 9.9）第 10 行中把它替代成无符号整数。

程序清单 9.8 Bits 的对象接口

```
/* bits.h: 大型位的置位 */
#ifndef BITS_H
#define BITS_H

#include <stdio.h>

typedef struct bits Bits;

Bits* bits_create(size_t nbits);
unsigned bits_to_uint(const Bits *);
Bits* bits_from_uint(Bits *, unsigned);
Bits* bits_set(Bits*, size_t bit);
Bits* bits_set_all(Bits* );
Bits* bits_reset(Bits*, size_t bit);
Bits* bits_reset_all(Bits*);
```

```

Bits* bits_toggle(Bits*, size_t bit);
Bits* bits_toggle_all(Bits *);
int bits_test(const Bits*, size_t bit);
int bits_any(const Bits*);
size_t bits_count(const Bits*);
Bits* bits_put(Bits*, FILE *);
Bits* bits_get(Bits*, FILE *);
void bits_destroy(Bits*);

#endif

```

Bits 对象由允许的位数、整数数组、数组中元素的个数和在置位操作后将未使用的位复位为 0 的屏蔽组成。程序清单 9.10 中测试程序表明在应用中该怎样使用 Bits 对象。

程序清单 9.9 Bits 对象的实现

```

/* bits.c */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include <assert.h>
#include "bits.h"

/* 选取整型的基 */
typedef unsigned int Block;          /* 第 10 行 */

/* 一些执行的细节 */
#define BLKSIZ      (CHAR_BIT * sizeof(Block))
#define offset(b)   (b % BLKSIZ)
#define mask1(b)    ((Block)1 << offset(b))
#define mask0(b)    (~mask1(b))

/* 数据结构 */
typedef struct bits
{
    size_t nbits_;      /* 位数 */
    Block *bits_;       /* 基数组 */
    size_t nblks_;      /* 基数组中块数 */
    Block clean_mask_;  /* 用来隐藏不常用的位 */
} Bits;

/* 私有函数 */
static size_t word_(const Bits* bp, size_t bit)
{
    return bp->nblks_ - 1 - bit/BLKSIZ;
}

```

```
static void set_(Bits* bp, size_t b)
{
    bp->bits_[word_(bp,b)] |= mask1(b);
}

static void reset_(Bits* bp, size_t b)
{
    bp->bits_[word_(bp,b)] &= mask0(b);
}

static void toggle_(Bits* bp, size_t b)
{
    bp->bits_[word_(bp,b)] ^= mask1(b);
}

static int test_(const Bits* bp, size_t b)
{
    return !(bp->bits_[word_(bp,b)] & mask1(b));
}

static size_t count_block_(Block n)
{
    size_t sum = 0;

    while (n)
    {
        if (n & (Block)1)
            ++sum;
        n >>= 1;
    }
    return sum;
}

static void cleanup_(Bits* bp)
{
    if (bp->nbits_ % BLKSIZ)
        bp->bits_[0] &= bp->clean_mask_;
}

/* 公有接口的执行在这里开始 */
Bits* bits_create(size_t nbits)
{
    Bits *bp = malloc(sizeof(Bits));
    size_t nbytes;

    if (bp == NULL)
```

```
        return NULL;

    /* 分配基数组 */
    bp->nblks_ = (nbits + BLKSIZ - 1) / BLKSIZ;
    nbytes = bp->nblks_ * sizeof(Block);
    bp->bits_ = malloc(nbytes);
    if (bp->bits_ == NULL)
    {
        free(bp);
        return NULL;
    }

    memset(bp->bits_, '\0', nbytes);
    bp->nbits_ = nbits;
    bp->clean_mask_ = ~(Block)0 >> (bp->nblks_*BLKSIZ - nbits);
    return bp;
}

unsigned bits_to_uint(const Bits* bp)
{
    size_t nblks = sizeof(unsigned) / sizeof(Block);
    if (nblks > 1)
    {
        int i;
        unsigned n = bp->bits_[bp->nblks_ - nblks];

        /* 集中 low-order 子块到一个无符号数 */
        if (nblks > bp->nblks_)
            nblks = bp->nblks_;
        while (--nblks)
            n = (n << BLKSIZ) | bp->bits_[bp->nblks_ - nblks];
        return n;
    }
    else
        return (unsigned) bp->bits_[bp->nblks_ - 1];
}

Bits* bits_from_uint(Bits* bp, unsigned n)
{
    size_t nblks = sizeof(unsigned) / sizeof(Block);
    assert(bp);
    memset(bp->bits_, '\0', bp->nblks_ * sizeof(Block));
    if (nblks > 1)
    {
        int i;
        if (nblks > bp->nblks_)
```

```
        nblks = bp->nblks_;
        for (i = 1; i <= nblks; ++i)
        {
            bp->bits_[bp->nblks_ - i] = (Block) n;
            n >>= BLKSIZ;
        }
    }
    else
        bp->bits_[bp->nblks_ - 1] = n;

    return bp;
}

Bits* bits_set(Bits* bp, size_t bit)
{
    assert(bp && (bit < bp->nbits_));
    set_(bp, bit);
    return bp;
}

Bits* bits_set_all(Bits* bp)
{
    assert(bp);
    memset(bp->bits_, ~0u, bp->nblks_*sizeof(Block));
    cleanup_(bp);
    return bp;
}

Bits* bits_reset(Bits* bp, size_t bit)
{
    assert(bp && (bit < bp->nbits_));
    reset_(bp, bit);
    return bp;
}

Bits* bits_reset_all(Bits* bp)
{
    assert(bp);
    memset(bp->bits_, '\0', bp->nblks_*sizeof(Block));
    return bp;
}

Bits* bits_toggle(Bits* bp, size_t bit)
{
    assert(bp && (bit < bp->nbits_));
    toggle_(bp, bit);
    return bp;
}
```

```
}

Bits* bits_toggle_all(Bits* bp)
{
    size_t nw;

    assert(bp);
    nw = bp->nblks_;
    while (nw--)
        bp->bits_[nw] = ~bp->bits_[nw];
    cleanup_(bp);
    return bp;
}

int bits_test(const Bits* bp, size_t bit)
{
    assert(bp && (bit < bp->nbits_));
    return test_(bp, bit);
}

int bits_any(const Bits* bp)
{
    int i;

    assert(bp);
    for (i = 0; i < bp->nblks_; ++i)
        if (bp->bits_[i])
            return 1;
    return 0;
}

size_t bits_count(const Bits* bp)
{
    int i;
    size_t sum;

    assert(bp);
    for (i = 0, sum = 0; i < bp->nblks_; ++i)
        sum += count_block_(bp->bits_[i]);
    return sum;
}

Bits* bits_put(Bits* bp, FILE *f)
{
    int i;
```

```

    assert(bp);
    for (i = 0; i < bp->nbits_; ++i)
        fprintf(f, "%d", bits_test(bp, bp->nbits_-1-i));
    return bp;
}

Bits* bits_get(Bits* bp, FILE *f)
{
    char *buf;
    char format[9];

    /*全部位复位 */
    assert(bp);
    bits_reset_all(bp);

    /* 分配字符串缓冲器 */
    buf = malloc(bp->nbits_+1);
    if (buf == NULL)
        return 0;

    /* 创建读格式(例如, " %16[01]") */
    sprintf(format, " %%%d[01]", bp->nbits_);
    if (fscanf(f, format, buf) == 1)
    {
        int i;
        size_t slen = strlen(buf);

        /* 在 bitset 中设置相应的位 */
        for (i = 0; i < slen; ++i)
            if (buf[slen-1-i] == '1')
                bits_set(bp, i);
    }
    free(buf);
    return bp;
}

void bits_destroy(Bits* bp)
{
    assert(bp);
    free(bp->bits_);
    free(bp);
}

```

程序清单 9.10 举例说明位对象

```

/* tbits.c: 测试位的接口 */
#include <stdio.h>

```

```
#include <assert.h>
#include "bits.h"

#define NBITS 36

main()
{
    int i;
    unsigned int n;
    Bits* bp = bits_create(NBITS);

    assert(bp);

    /* 将偶数位置位 */
    for (i = 0; i < NBITS; i += 2)
        bits_set(bp, i);
    bits_put(bp, stdout);
    printf(" (%d)\n", bits_count(bp));

    /* 高位取反 (上面的一半) */
    for (i = NBITS/2; i < NBITS; ++i)
        bits_toggle(bp, i);
    bits_put(bp, stdout);
    printf(" (%d)\n", bits_count(bp));

    /* 复位低位 (下面的一半) */
    for (i = 0; i < NBITS/2; ++i)
        bits_reset(bp, i);
    bits_put(bp, stdout);
    printf(" (%d)\n", bits_count(bp));

    /* 读一个位字符串 */
    fputs("Enter a bit string: ", stderr);
    bits_put(bits_get(bp, stdin), stdout);
    printf(" (%d)\n", bits_count(bp));

    /* 与无符号数之间转换 */
    n = bits_to_uint(bp);
    printf("n: %u\n", n);
    bp = bits_from_uint(bp, n);
    bits_put(bp, stdout);
    printf(" (%d)\n", bits_count(bp));

    /* 测试 any() 和 test() */
    printf("any? %d\n", bits_any(bp));
    printf("test(0)? %d\n", bits_test(bp, 0));
}
```



```

/* 取反并复位 */
bits_put(bits_toggle_all(bp), stdout);
printf(" (%d)\n", bits_count(bp));
bits_put(bits_reset_all(bp), stdout);
printf(" (%d)\n", bits_count(bp));
bits_put(bits_set_all(bp), stdout);
printf(" (%d)\n", bits_count(bp));

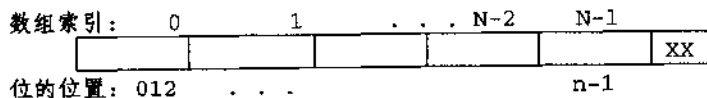
bits_destroy(bp);
return 0;
}

/* 示例执行结果:
01010101010101010101010101010101 (18)
10101010101010101001010101010101 (18)
10101010101010101000000000000000 (9)
Enter a bit string: 1010010001000010000001000000100000001
1010010001000010000001000000100000001 (8)
n: 1142980865
0000010001000010000001000000100000001 (6) /* The lower 32 bits remain */
any? 1
test(0)? 1
111110111011110111110111110111110111110 (30)
000000000000000000000000000000000000 (0)
111111111111111111111111111111111111 (36)
*/

```

9.4 位字符串

位对象中存储方案把 0 位放在数组的最后, 因此位是从右到左编号的, 就像它们在单个整数里一样。在没有置位的数字应用的情况下, 比如上面选取表的例子, 像字符串一样从左往右计位可能会更自然一些。对 `bits.c` 做方向上的改变是很微不足道的。



第 `b` 位的字块现在是:

`B = b/BLKSIZ`

并且偏移是从左计算而不是从字块的右面开始的:

`offset = BLKSIZ - b%BLKSIZ - 1`

9.5 Wish List

应该为这些对象增加许多特征, 这是能够做到的。提供应用于整个对象的按位运算符,

将会很方便，例如：

```
Bits *bp1, *bp2, *bp3;
/* ... */
bits_and(bp3, bp1, bp2);
bits_shift_left(bp3, 4);
```

对于位字符串来说，像字符串一样让它们变大变小是很自然的。这里可借助于 C++ 标准库。C++ 标准库提供了置位（bitset）类模板，以及带有从左到右位字符串的 `vector<bool>`，它是 `vector` 类模板的特化。

9.6 bitset 模板

C++ 标准库定义了一个置位类模板，其功能很像上面的结构位（`struct bits`），但是它经过改进从 C++ 中得到了更多方便。它还是库中唯一一个带有无类型模板参数的模板，该模板代表了它处理的固定位数。它的定义从如下开始：

```
template<size_t N>
class bitset
{
    //等等。
};
```

因为在模板实例化时位数对于编译器是可用的，并且基本缓冲区可在堆栈中分配，所以没有必要像 `struct bits` 那样使用灵活的存储方式。就像程序清单 9.11 所示，`bitset` 支持所有的一般位运算符。它还提供了运算符[]，以便你能用数组符号存取个别位，例如：

```
if (b[i])
```

或

```
b[i] = false;
```

程序清单 9.11 测试 bitset 类模板

```
// tbitset.cpp: 测试 bitset 类模板
#include <iostream>
#include <bitset>
#include <string>
using namespace std;

const size_t NBITS = 36;
void print(const bitset<NBITS>&);

main()
{
    bitset<NBITS> b;
    cout.setf(ios::boolalpha);

    //偶数位置位:
```

```

    for (int i = 0; i < b.size(); i += 2)
        b.set(i);
    print(b);

    /* 高位取反 (上面的一半) */
    for (int i = b.size()/2; i < b.size(); ++i)
        b.flip(i);
    print(b);
    // 复位低位 (下面的一半):
    for (int i = 0; i < b.size()/2; ++i)
        b.reset(i);
    print(b);

    // 读一个位字符串:
    cout << "Enter a bit string: ";
    string s;
    cin >> s;
    bitset<NBITS> b2(s);
    print(b2);

    /* 测试 any() 和 test() */
    cout << "any? " << b2.any() << endl;
    cout << "test(0)? " << b2.test(0) << endl;

    // 取反、复位和置位:
    b2.flip();
    print(b2);
    b2.reset();
    print(b2);
    b2.set();
    print(b2);
}

void print(const bitset<NBITS>& b)
{
    cout << b.to_string() << " (" << b.count() << ")\n";
}

//输出:
01010101010101010101010101010101 (18)
10101010101010101001010101010101 (18)
10101010101010101000000000000000 (9)
Enter a bit string: 1010010001000010000001000000100000001
101001000100001000000100000010000001 (8)
any? true

```

```

test(0)? true
0101101110111101111101111101111110 (28)
00000000000000000000000000000000 (0)
1111111111111111111111111111111111 (36)

```

9.7 vector<bool> 模板特化

该库还提供了向量容器的专门版本，该容器经过优化后用来存储位。正如所预想的那样，它有从左到右的定向，因此被认为是一个动态分配字节长度的位字符串。除了常见的向量操作外，它还提供了 flip 成员函数，该函数反置所有位 (b.flip) 或个别位 (b[i].flip)，其他按位运算用 vector<bool> 无法实现。参见程序清单 9.12 中的程序示例。

程序清单 9.12

```

// tbitvec.cpp: 测试向量<bool>
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

void print_alpha(const vector<bool>&);
void print_digits(const vector<bool>&);

main()
{
    vector<bool> b;

    // 增加
    b.resize(4);           // "0000"
    b[0] = 1;              // "1000"
    b[3] = true;           // "1001"

    // 按布尔值打印
    print_alpha(b);

    // 用数字打印
    print_digits(b);

    // 在前面插入更多的(以 LIFO 顺序):
    for (int i = 0; i < 5; ++i)
        b.insert(b.begin(), bool(i % 2));
    print_digits(b);

    // 取反位:

```

```

        b.flip();
        print_digits(b);
        // 单独位的存取:
        b[1] = false;
        b[2] = 0;
        b[3].flip();
        print_digits(b);

        // 调整大小:
        b.resize(5);
        print_digits(b);
    }

void print_alpha(const vector<bool>& b)
{
    cout.setf(ios::boolalpha);
    copy(b.begin(), b.end(), ostream_iterator<bool>(cout, ""));
    cout<<endl;
}

void print_digits(const vector<bool>& b)
{
    cout.unsetf(ios::boolalpha);
    copy(b.begin(), b.end(), ostream_iterator<bool>(cout, ""));
    cout<<endl;
}

// Output:
true false false true
1001
010101001
101010110
100110110
10011

```

9.8 小结

- 位操作支持系统编程而且能降低内存需要。
- 按位运算符允许访问整数位的子集（但是要注意运算符的优先权）。
- `bitset` 类模板支持有效的固定大小的位设置操作。
- `vector<bool>` 模板规范支持动态大小的位字符串。

类型转换和强制类型转换

由于C是为了系统编程而发明的，因此它有一些“接近机器”的特性。这些特性包括自增和自减运算符（它们通常映射为一个单独的机器指令）、位段操作、指针和地址运算符以及丰富的低级数据类型。其他什么语言能有8种风格的整数呢？对C/C++程序员来说，有必要知道不同类型对象之间是如何相互作用的。当不同数据类型混合使用时，如在数学表达式中，或当一个对象被赋值或当作参数被传递给另一种类型的对象时，编译器隐式地将对象从一种类型转换成另一种类型。虽然类型转换可能在有些时候使对象变宽或变窄，但在大多数情况下，类型转换由重新解释基本的位模式来实现的，而不是真正地改变类型（注意：除非特殊说明，在本章中所有的例子都用32位的整数运算）。

10.1 整数的升级

在短整型、整型和长整型大小都不相同的环境中，根据每种类型所能表示的最大值，8种整数类型可排序如下：

```
signed char < unsigned char < signed short < unsigned short  
               < unsigned int < unsigned int < signed long  
               < unsigned long
```

然而在大多数环境中，整型不是短整型就是长整型。在任何情况下，整型表达式中只能用到短整型和长整型对象或者它们的无符号型变量。当一个窄于整型（int）的整数类型对象在表达式中使用时，它被升级为整型（如果需要也可以升级为无符号整型）来计算。例如，考虑下面的程序：

```
/*char1.c*/  
#include <stdio.h>  
main()
```

```

{
    char c1=100;
    char c2=200;
    char c3=c1+c2;
    printf("c1==%d\n",c1);
    printf("c2==%d\n",c2);
    printf("c3==%d\n",c3);
}

```

//输出:

c1== 100

c2== -56

c3== 44

在字符型占 8 位的环境里, 它将打印值 44。这个程序出现了有趣的特征, 如果你问 c2 是怎么回事? 为什么 c2 输出是 -56? 因为值 200 超出了有符号字符所能表示的范围, 所以 c2 被解释成 -56。但是还有更多需要考虑的东西。除了 char* 格式说明外, printf 不对参数进行类型检查, 因此 c2 在 printf 看到它之前被升级为整型。编译器可以自由地把字符型解释成有符号型字符或是无符号型字符。正如所看到的, 这些例子中我用到的编译器采用了前者的形式。标准相容的编译器通过符号扩展把有符号字符升级为整型, 这意味着它用和原来字符符号位的相同值来迁移新的高位。以下是以十六进制改变打印语句时所输出的结果:

c1== 0x64

c2== 0xFFFFF8

c3== 0x2C

值为 200 (0xc8) 的字符含有最有意义的位集合, 因此它升级到的整型值为 0xFFFFF8,

然后, 程序这样计算总和

00000064

+ FFFFFFF8

= (1) 0000002C

它舍去高位变成 0x2C 或者 44。

把变量声明成无符号字符型则输出变为:

//十进制

c1==100

c2== -56

c3==44

//十六进制:

c1== 0x64

c2== 0xC8

c3== 0x2C

总和 300 超出了 1 个字节所能容纳的最大值, 结果是溢出的数值, 它可用求余操作来表达:

44== 300 % 256

这里 256== UCHAR_MAX+1 (UCHAR_MAX 在 <limits.h> 中定义)。正如所看到的, 无

符号字符通过在高位上补 0 来完成升级。在这种情况下程序计算总和为：

```
00000064
+ 000000C8
-----
= 0000012C
```

它舍去高位变成 0x2C，又一次是 44。

无符号量和有符号量升级时的区别是很有意义的，例如在程序清单 10.1 中的“转储”程序，用两个十六进制数输出输入文件的每一个字节，每行 16 位，后面跟着相应的 ASCII 表示（见程序清单 10.6 的输出结果）。如果数组 buf 没有被声明为无符号型，任何字节值大于或等于 128 的数在显示时将加上额外的位，假设额外的位为 FFFFFFFF，这使输出结果偏离了实际值（重新编译亲自看看结果）。

程序清单 10.1 十六进制/ASCII 转储程序

```
/* dump.c: 以十六进制和 ASCII 形式显示文件的字节 */
#include <stdio.h>
#include <ctype.h>

const int NBYTES = 16;
void dump(FILE *, char *);

main(int argc, char *argv[])
{
    /* 在命令行处理文件 */
    for (int i = 1; i < argc; ++i)
    {
        FILE *f;

        if ((f = fopen(argv[i], "rb")) == 0)
            fprintf(stderr, "Can't open %s\n");
        else
        {
            dump(f, argv[i]);
            fclose(f);
            putchar('\\f');
        }
    }
    return 0;
}

void dump(FILE* f, char* s)
{
    unsigned char buf[NBYTES];
    int count;
    long size = 0L;
```

```

printf("Dump of %s:\n\n",s);
while ((count = fread(buf,1,NBYTES,f)) > 0)
{
    /* 打印字节计数器 */
    printf(" %06X ",size += count);

    /* 打印十六进制字节 */
    for (int i = 0; i < NBYTES; ++i)
    {
        /* 在列之间打印装订线间距 */
        if (i == NBYTES / 2)
            putchar(' ');

        /* 显示十六进制字节 */
        if (i < count)
        {
            printf(" %02X",buf[i]);
            if (!isprint(buf[i]))
                buf[i] = '.';
        }
        else
        {
            /* 为最后一行的部分留间隔 */
            fputc(" ",stdout);
            buf[i] = ' ';
        }
    }

    /* 打印文本字节 */
    printf(" |%16.16s|\n",buf);
}
}

```

练习

测试程序清单 10.3 中程序的输出。赋值语句

`y=x+x;`

两端的输出不相同，为什么？

程序清单 10.2 转储命令 dump.exe 的部分输出

```

000010 4D 5A 50 00 02 00 00 00 04 00 0F 00 FF FF 00 00 |MZP.....|
000020 B8 00 00 00 00 00 00 00 40 00 1A 00 00 00 00 |.....@.....|
000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000040 00 00 00 00 00 00 00 00 00 00 00 00 01 00 00 |.....|
000050 BA 10 00 0E 1F B4 09 CD 21 B8 01 4C CD 21 90 90 |.....!..L!..|

```

```
000060 54 68 69 73 20 70 72 6F 67 72 61 6D 20 6D 75 73 |This program mus|
000070 74 20 62 65 20 72 75 6E 20 75 6E 64 65 72 20 57 |t be run under W|
000080 69 6E 33 32 0D 0A 24 37 00 00 00 00 00 00 00 |in32..$7.....|
```

```
//等等...
```

程序清单 10.3 整型升级练习

```
// promote3.cpp
#include <stdio.h>

main()
{
    char x = 124;
    char y = x + x;

    printf("x == %02X\n",x);
    printf("x + x == %02X\n",x+x);
    printf("y == %02X\n\n",y);
}

//输出:
x == 7C
x + x == F8
y == FFFFFFF8
```

10.2 降级

将整数转换成窄的无符号数的结果是此整数除以 2^n 后的余数，这里 n 是无符号数所占的位数，这就解释了 10.1 节的 `char1.c` 中为什么 300 转换成无符号字符型的结果会是 44。这种转换叫做降级，也叫做变窄转换。然而，如果相对的窄类型是有符号的，并且小类型不能表示大类型的值，则结果是由实现定义的。如程序清单 10.4 所示，如果需要的话，我的 16 位编译器舍去高位并根据两个数的反码来解释这个值。例如，60000U 的位模式是：

```
1110 1010 0110 0000 .
```

程序清单 10.4 说明变窄转换（16 位版本）

```
/* narrow.c */
#include <stdio.h>

void f(int);
void g(short);

main()
{
    unsigned n = 60000U;
```

```
    long m = 70000L;
    float x = 1.23e10;

    f(n);
    f(x);
    g(m);
    return 0;
}

void f(int i)
{
    printf("%d\n", i);
}

void g(short i)
{
    printf("%hd\n", i);
}

/* 输出:1
-5536
7168
4464
*/
```

为了把它解释成负整数，两种类型的互补规则对这些位取反并加 1 来求值：

```
    0001 0101 1001 1111
+                               1
-----
= 0001 0101 1010 0000
```

这个值是 5536，因此 (int) 60000U == -5536。

70000L 的位模式是：

```
0000 0000 0000 0001 0001 0001 0111 0000
```

转换成短整型（在 16 位环境中和整型相同）时简单地舍去高 16 位，剩下的值是 4464。

将浮点数转换成整数的时候舍去了小数部分，但只是在实现定义的模式下是这样。当浮点值太大而不能适合整数时，结果是不确定的。

10.3 算术类型转换

运用浮点数就像搬沙子一样——每次处理了一部分同时也失去了一部分。大多数实数和大的

¹ 译注：这是 16 位机的输出结果。32 位机的输出结果为：

```
60000
-584901632
4464
```

整数不能在有限的浮点数字系统中表示，在计算中用最接近它的值来代替它。程序清单 10.5 说明了这个事实。从有限精度运算继承来的舍入误差导致了 100×23.4 的结果接近 2339，而不是 2340。

在标准的 C/C++ 中有三个级别的浮点数精确度定义：单精度、双精度和长双精度。用单精度表示的值是用双精度表示的值的一个子集，用双精度表示的值按顺序是长双精度表示的值的子集。未加修饰的常量，如 100.0，是双精度类型的。为了强调常量是另一种浮点类型，可以用一个合适的后缀，如表 10.1 所示。

程序清单 10.5 举例说明浮点运算的不精确度

```
// float.cpp
#include <stdio.h>
main()
{
    int i = 100;

    i *= 23.4;
    printf("%d\n", i);
    printf("100 * 23.4 %s 2340.0\n",
        (100 * 23.4 == 2340.0) ? "==" : "!=");
}

//输出:
2339
100 * 23.4 != 2340.0
```

表 10.1 浮点类型的后缀和格式描述符

类 型	后 缀	例 子	格式说明符
单精度	f 或 F	100.0f, 100.0F	%f
双精度	没有	100.0	%f
长双精度	l 或者 L	100.0l, 100.0L	%Lf

当把一个浮点数降级为另一个具有小精度的浮点类型时，像从双精度到单精度，有三种可能的输出：

1. 被降级的原始数超出了目标类型的范围。在这种情况下，结果是不确定的。
2. 原始数在目标类型的范围之内但不能用目标类型表示。结果是最接近的偏大值或偏小值，这由所用的舍入算法决定（它是由实现定义的）的。
3. 原始数在目标精确度内能准确地表达，在这种情况下没有精度损失。

在程序清单 10.6 中的程序说明了 `ULONG_MAX`，它是最大的无符号长整型（在我的平台上是 4 294 967 295），在我的编译器中不能用单精度表示它，但是可以表示 `ULONG_MAX+1`（因为它是 2 的幂，而浮点数字系统是以二进制为基础的）。然而，双精度和长双精度都有足够的精度来表示 `ULONG_MAX`。

程序清单 10.6 举例说明三种浮点类型

```
// float2.cpp
```

```
#include <stdio.h>
#include <limits.h>

main()
{
    float x = ULONG_MAX;
    double y = ULONG_MAX;
    long double z = ULONG_MAX;

    printf("%f\n", x);
    printf("%f\n", y);
    printf("%Lf\n", z);
}
```

//输出:

```
4294967296.000000
4294967295.000000
4294967295.000000
```

当两个不同类型的数作为二进制算术运算的操作数出现时,表达式通过把一种类型转换成另一种类型,或者在整数很小的情况下通过普通的整数升级来达到平衡。如果两个操作数中有浮点类型,则相对窄的对象被转换成最大的浮点类型。如果两个实参都是整数,则它们经过整数升级得到的操作数是临时性的,而且不是整型就是长整型(或在适当的时候是它们的无符号型式)。这样为了运算,相对窄的类型被转换成宽类型。换句话说,在二进制数字运算中用到的类型根据长整型和整型的关系按照下面的一个优先级列表向上升级:

```
if sizeof (long )> sizeof (int):
```

```
    long double
    double
    Float
    unsigned long
    long
    unsigned int
    Int
```

```
if sizeof (long )== sizeof (int):
```

```
    long double
    double
    Float
    unsigned long == unsigned int
    Long == int
```

10.4 函数原型

函数原型所做的不只是在编译期对函数实参进行类型检查。它们也提供编译器惯用的将

实参自动地转换成相应的形参类型的信息。例如，在旧式的函数定义中（即没有函数原型）不被发现的常见错误是把一个整数实参传递给双精度形参：

```
f (1);          /*一个大麻烦！函数 f 要求一个双精度参数*/
...
void f(x)
double x;
{
...

```

如程序清单 10.7 所示，这里提供了完整的函数原型声明，从而在使用它之前就把整型隐式转换成双精度型。同时它也说明了，在这种从双精度型转换成整型的情况下，窄化转换是很危险的（因为 1.0e6 的整型表示不能用 16 位短整型表达）。

程序清单 10.7 揭示通过函数原型完成的自动类型转换

```
/* proto.c */
#include <stdio.h>

void f2(short,double);

main()
{
    f1(1,3);
    f1(1.0, 3.0);

    f2(1,3);
    f2(1.0,3.0);
    f2(1.0e6,3.0e6);
}

f1(x,y)          /*旧式的函数声明*/
short x;
double y;
{
    printf("%d, %f\n",x,y);
}

void f2(short x, double y)
{
    printf("%d, %f\n",x,y);
}

/* 输出:
1, 0.000000
0, 0.000000
1, 3.000000
1, 3.000000

```

```
16960, 3000000.000000
```

```
*/
```

你应对不进行参数检查负责。例如，在下面 `printf` 的原型中，省略号表明了第一个形参之后可以跟任何类型任何数量的实参。

```
printf(const char*,...);
```

由于编译器无法知道这些缺省参数应该是什么类型的，因此它无法发现下面的错误：

```
#include <stdio.h>
```

```
main()
```

```
{
```

```
    int x=1;
```

```
    printf("%d\n",x);    /*不匹配! */
```

```
}
```

```
//输出:
```

```
65537
```

编译器对没进行检查的实参只能进行默认的升级，这包括整数的升级和从单精度到双精度的类型转换。

10.5 显式类型转换

目前为止我们所讨论的类型转换都是隐式类型转换，它们在你没有做任何干涉的情况下发生。可以在任何时候用强制类型转换操作符显式地要求类型转换。例如，在下面的赋值语句中，`x` 的小数部分没有用。

```
int i;
```

```
double x;
```

```
/*...*/
```

```
i = i + x;
```

这里程序的执行顺序是：

1. 把 `i` 转换成双精度。
2. 用双精度数学运算把 `i`（双精度）和 `x` 相加。
3. 把结果舍位变成整数并把它值赋给 `i`。

如果正在计算十亿分之一秒，可以通过强制类型将 `x` 转换成整型从而避免双精度算术运算：

```
i = i + int(x);
```

当对指针进行处理时强制类型转换是很方便的。例如，如果需要检查一个整数的各个字节，可以这样做：

```
int i;
```

```
char *cp=(char*) &i;
```

```
char hi_byte = *cp++;
```

```
char lo_byte = *cp;
```

```
/*等等。*/
```


为了让编译器知道你清楚自己在做什么采用强制类型转换是很有必要的，因为正常情况下不允许出现指向不同类型的混合指针。当在 C 或 C++ 中用 `void*` 的返回值给指针赋值时，就没有必要用强制类型转换。但在 C++ 中，给 `void` 型指针赋值时不用强制类型转换则是一个错误。

10.6 函数风格强制类型转换

既然强制类型转换是一个在 C++ 类型系统中常用到的显式转换，因此你应该谨慎地使用强制类型转换。一些人争论说 C 的强制类型转换语法格式很难看，因此 C++ 允许函数风格的强制类型转换。例如可以将下面的语句：

```
i=i + (int)x;
```

重写成：

```
i=i + int(x);
```

这样除了看起来舒服以外，这个符号也与用来创建临时变量和初始化对象的构造函数的用法相一致，如下面所示：

```
#include "foo.h"      //定义 Foo 类
Foo f=Foo(1) + Foo(2);
```

在这里，类 `Foo` 必须有一个只带一个整型参数的构造函数。更多的细节请参见下面的“用户定义类型转换”部分。老式风格的强制类型转换形式 `(T) X` 依然有效，但不推荐使用。

函数风格的强制类型转换的一个缺点是只可以使用单一符号的类型名。例如，下面的语句是非法的：

```
const unsigned char*p= const unsigned char *(p);
```

解决这个问题的方法是采用 `typedef`：

```
typedef unsigned char *const_ucharp;
const unsigned char*p= const_ucharp(p);
```

由于强制类型转换存在着潜在的危险，可能用难看的语法格式会好些，因为它不同于正常的代码。现在首选的技术是使用 C++ 新风格的强制类型转换，这将在下面讨论。

10.7 const 的正确性

`const` 限定符是 C++ 类型系统的一个关键要素。例如，当函数参数中有一个只读的指针或引用时，应该把它声明为常量 (`const`)。考虑 `memcpy` 函数的原型：

```
void *memcpy(void *to, const void *from, size_tn);
```

指向源缓冲区 `from` 的指针被声明为指向常量的指针，这意味着这个函数不想改变指针所指向的内容。这不仅防止了函数内部无意中的赋值，而且还允许把指针作为实参传递给常量。如果在第二个形参中没有 `const` 限定符，则在下面的语句中不能传递 `t`：

```
char*s= ...;
const char *t=...;
memcpy(s, t,10);
```

编译器需要保证函数 `memcpy` 不能试图改变 `t` 所指向的内容。在原型中用 `const`，指向常量和非常量的指针都能传递给 `t`。

以上所述既适用于指针，也适用于引用。在 C++ 中按值传递对象的习惯用法是传递常引用：

```
void g(const Foo& f);
```

这样保证了 `g` 不能改变 `f`（不管怎样，没有用强制类型转换）。

可以通过把 `const` 放在星号的后面来声明一个本身不可以改变的指针：

```
char *const p;
```

```
*p = 'a';           //OK，只有指针是常量
```

```
++p;                //错误，不能改变指针
```

为了禁止改变指针及其所引用的内容，在两个位置都用 `const`：

```
const char * const p;
```

```
char c;              //OK—可以读内容
```

```
*p = 'a';            //错误
```

```
++p;                 //错误
```

在 C++ 类中的常量成员函数不允许改变使用它的对象。例如，一个 `Date` 类可能有单独的函数来检查和改变月份：

```
class Date
```

```
{
```

```
    int month_;
```

```
    //余下的实现省略了
```

```
public:
```

```
    int month() const;    //返回月
```

```
    void month(int);      //改变月
```

```
    //等等。
```

```
};
```

函数 `month` 是常量，因为它仅返回当前的值。因此，下面的两个函数调用都是有效的：

```
const Date d1;
```

```
Date d2;
```

```
d1.month();
```

```
d2.month();
```

但是，不能改变 `d1` 的月份，如下：

```
d1.month(10);
```

因为不能对一个常量对象调用非常量成员函数。编译器通过下面的和对对象有联系的隐藏的 `this` 指针来检测错误。当对象是非常量时，`this` 指针的类型是 `Date*const`，这意味着“`this` 是一个指向 `Date` 的常指针”。因此不能改变 `this` 指针（无论如何，这可能是不可思议的），但是能改变它所指向的内容；可以在非常量成员函数中改变它的私有数据成员。从另一方面来说，如果隐含的对象是常量，则 `this` 指针有如下的声明：

```
const Date* const this;
```

它禁止改变任何成员。常量成员函数希望有这种 `this` 指针，因此如果调用非常量的对象，将会出现不匹配，编译器将报错。

然而，有些时候可能想在一个常量成员函数中改变数据成员。例如，函数不改变任何用

户知道的东西，但是可能有一些便于更新的私有声明变量（如为了效率而隐藏的值）。你仍然想让用户可以调用常量成员的函数，而他/她并不必知道你的优化秘密。这种 `const` 概念的使用有些时候叫做语义常量（相对于位逻辑常量）。例如，考虑下面的类定义：

```
class B
{
public:
    void f () const;
private:
    int state;
};
```

为了允许 `B::f` 改变 `state`，并可以被常量对象调用，有两种选择。第一，可以在 `f` 的实现里从 `this` 指针中“去掉 `const`”：

```
void B::f() const
{
    ( (B*)this)->state = ...; //在这里改变是合法的
    ...
}
```

记住在 `f` 中 `this` 的类型是 `const B* const`。表达式 `(B*) this` 临时地中止 `const` 的作用，允许用户进行赋值。去掉 `const` 的更好办法是用新风格的强制类型转换中的一个：

```
const_cast<B*>(this)->state = ...; //在这里改变是合法的
```

第二种（非常好的）选择是把 `state` 声明为可变的数据成员，如下：

```
class B
{
public:
    void f() const;
private:
    mutable int state;
};
```

这样就允许不用强制类型转换就可以改变 `state`，即使它是个常量。

10.8 用户定义的类型转换

带有单参数构造函数的类定义了从实参类型到类类型的隐式转换。例如，在程序清单 10.8~10.10 中出现的有理数类有下面的构造函数：

```
Rational (int n=0, int d=1);
```

它在初始化一个有理数时允许带有 0 个、1 个或 2 个参数。任何时候当一个整数出现在带有一个 `Rational` 对象的表达式时，整数被隐式地转换成分母是 1 的 `Rational`。这使得下面这样的语句有意义了：

```
Rational y(4);
cout<< 2 / y << endl;
```

当编译器看到表达式：

2 / y

时，编译器用标记 `operator/ (int, Rational)` 寻找一个全局函数。如果没有找到，除了内置的 `operator/ (int, int)` 以外，它注意到存在函数 `operator/ (const Rational&, const Rational&)`。因此它寻找从 `int` 到 `Rational` 或者从 `Rational` 到 `int` 的类型转换。如果两种转换都存在（在这里它们不存在），编译器将报错，因为它不知道去选择哪一个转换。因为 `Rational` 类允许带有单个参数的构造函数，编译器用这个构造函数将 2 转换成 `Rational (2)`，然后调用 `operator/ (Rational (2), y)`。

程序清单 10.8 小数运算类的接口和内联函数

```
// rational.h

#include <iosfwd>          //包含关键字的输入输出流声明
                        //（系统开销比<iostream>要少）
using namespace std;

class Rational
{
public:
    Rational(int n = 0, int d = 1);

    // 有理数的操作
    friend Rational operator+(const Rational&, const Rational&);
    friend Rational operator-(const Rational&, const Rational&);
    friend Rational operator*(const Rational&, const Rational&);
    friend Rational operator/(const Rational&, const Rational&);
    Rational operator-() const;

    // I/O 操作符
    friend ostream& operator<<(ostream&, const Rational&);
    friend istream& operator>>(istream&, Rational&);

private:
    int num;
    int den;
    void reduce();
    static int gcd(int, int);
};

inline Rational::Rational(int n, int d)
{
    num = n;
    den = d;
    reduce();
}
```

```

inline Rational operator+(const Rational& a, const Rational& b)
{
    Rational r(a.num*b.den + b.num*a.den, a.den*b.den);
    r.reduce();
    return r;
}

inline Rational operator-(const Rational& a, const Rational& b)
{
    Rational r(a.num*b.den - b.num*a.den, a.den*b.den);
    r.reduce();
    return r;
}

inline Rational operator*(const Rational& a, const Rational& b)
{
    Rational r(a.num*b.num, a.den*b.den);
    r.reduce();
    return r;
}

inline Rational operator/(const Rational& a, const Rational& b)
{
    Rational r(a.num*b.den, a.den*b.num);
    r.reduce();
    return r;
}

inline Rational Rational::operator-() const
{
    return Rational(-num,den);
}

```

程序清单 10.9 测试有理数 Rational 类

```

// trat.cpp
#include <iostream>
#include "rational.h"
using namespace std;

main()
{
    Rational x, y(4), z(5,7);

    x = 3;
    cout << "x == " << x << endl;
}

```

```
    cout << "y == " << y << endl;
    cout << "z == " << z << endl;

    cout << "x + y == " << x + y << endl;
    cout << "y - z == " << y - z << endl;
    cout << "x * 2 == " << x * 2 << endl;
    cout << "2 / y == " << 2 / y << endl;
}
```

//输出:

```
x == 3 / 1
y == 4 / 1
z == 5 / 7
x + y == 7 / 1
y - z == 23 / 7
x * 2 == 6 / 1
2 / y == 1 / 2
```

程序清单 10.10 Rational 类的实现

```
// rational.cpp
#include <iostream>
#include "rational.h"
using namespace std;

// 求两个整型数的最大公约数(欧几里德(Euclid)算法)
int Rational::gcd(int x, int y)
{
    while (y != 0)
    {
        int rem = x % y;
        x = y;
        y = rem;
    }
    return x;
}

// 减少分数的最少项
void Rational::reduce()
{
    int g = gcd(num, den);

    if (g > 1)
    {
        num /= g;
        den /= g;
    }
}
```

```

    }
}

istream& operator>>(istream& is, Rational& r)
{
    char c;

    is >> r.num >> c;
    if (c == '/')
        is >> r.den;
    else
    {
        r.den = 1;
        is.putback(c);
    }
    r.reduce();
    return is;
}

ostream& operator<<(ostream& os, const Rational& r)
{
    os << r.num << " / " << r.den;
    return os;
}

```

`operator/(const Rational&, const Rational&)` 是全局函数, 这一点很重要。如果我把 `operator/` 定义为成员函数, 如下所示:

```
Rational Rational::operator/ (const Rational&);
```

这样编译器将可以求表达式 $y/2$ 的值, 而不是 $2/y$ 的值, 因为成员操作符函数要求左操作数是它们类的对象。然而, 可以提供服务于所有可能的操作数相结合的函数, 例如:

```
//下面这些分别处理 r/r, r/i, 和 i/r.
```

```
Rational Rational::operator/ (const Rational&);
```

```
Rational Rational::operator/ (int);
```

```
Rational operator/ (int, const Rational&);
```

虽然这种策略不需要从 `int` 到 `Rational` 的任何类型转换, 但是这样做会使所创建的类有大量的操作符而显得太冗长了。结合适当的构造函数来定义全局操作符函数允许对称的、混合模式的并且有最小麻烦的二进制操作。

虽然在这里不想这样做 (因为上面所提的很含糊), 但是怎样定义从 `Rational` 到 `int` 的隐式转换呢? 没有整型类定义可供你在任何需要将一种新类型转换成整型的时候增加具有单独参数的构造函数。在类本身的定义中需要一些方法提供从类到其他类型的转换, 甚至是内置的类型。可以用类型转换操作符来做这件事情。为了将类的对象转换成 `T` 类型, 仅仅需要把下面的函数定义成类的成员:

```
operator T() const;
```

如果觉得必须从 `Rational` 类型转换成 `int`，可以用成员函数来做一个显式转换，如下所示：

```
int to_int() const {return num/den;}
```

并且在需要的时候调用它，如下

```
int i = r.to_int();
```

一些实现提供了一个字符串类，这个类包括 `char*` 转换操作符，它允许用字符串对象作为参数传递给一个需要 `char*` 参数的 ANSI-C 字符串函数，该函数需要一个 `char*` 类型参数；例如，

```
string s;
strcpy (s, "hello");
```

在程序清单 10.11 中的程序说明了仍然需要小心地使用其函数带有未检查的参数的类型转换。表达式

```
(char*) s
```

调用 `string::operator char*` 并像所希望的那样返回指向 `data` 类型的指针。但是由于 `printf` 的第二个参数和后面的参数没有进行类型检查，第二个 `printf` 的调用中没有发生隐式类型转换。

如果有一个只带单个参数的构造函数的类，但是想禁止使用隐式转换，那么可用 `explicit` 限定符来使隐式类型转换无效。标准 C++ 容器显式地声明了所有带单个参数的构造函数。例如，构造函数

```
explicit vector<T>::vector (size_t n, const T& = T() )
```

允许用 `n` 的默认值初始化一个向量，如

```
vector<string> v(5); // 始于 5 的空字符串
```

程序清单 10.11 说明不进行类型检查的参数不能进行转换

```
// uncheck.cpp
#include <stdio.h>
#include <string.h>

class string
{
public:
    string(char *s)
    {strcpy(data = new char[strlen(s)+1], s);}
    operator char*()
    {return data;}
private:
    int dummy;
    char *data;
};

main()
{
    string s = "hello";
    printf("%s\n", (char*) s);
    printf("%s\n", s);
}
```



```

}

//输出:
hello
(null)
没有 explicit 限定符, 语句
v=10;

```

将用默认元素 10 来创建一个临时的向量并把它赋给 v, 这是非常奇怪的。关键词 `explicit` 只能改变构造函数的声明。

10.9 加强运算符[]

一个典型的字符串类的实现包括带下面标记的成员函数:

```

char& opertor[] (size_t);
char operator[] (size_t) const;
第一种方式的实现通常像这样:
char& string::operator [] (size_t pos)
{
    return data[pos];
}

```

这里 `data` 是基本的字符数组。这个成员函数提供了典型的下标行为, 所有程序员在编程时都要用到它:

```

string s;
char c = s [0];
因为 string::operator [] 返回一个引用, 它可以被当作左值使用, 如下面的赋值语句:
s [0] = 'a';

```

需要发生不同的行为时情况将会怎样? 这由下标是在赋值语句的左边还是右边决定。在标准 C++ 库中的 `bitset` 类模板就是这样。例如, 表达式

```

bitset<16> b;
b[0] = 1;

```

想给第 0 位置位, 但是下面的表达式测试第 0 位:

```

int x = b[0];

```

由于这两种操作在任何合理的实现中截然不同, 所以下标出现在赋值语句的左边还是出现在赋值语句的右边是需要我们用一些办法来区分的。解决的办法就是引进一个新的类——叫做 `reference`, 它只有两个公有成员函数: 一个赋值操作符和一个隐式整数类型转换 (见程序清单 10.12)。我已把类 `reference` 嵌套在样本类 `Foo` 中, 私有地保存了它将索引的 `Foo` 对象的引用以及索引本身的位置。构造函数是私有的, 因此只有 `Foo` 类型的对象能创建临时的 `reference` 对象 (`Foo` 是 `reference` 的友元)。

函数 `Foo::operator[]` 只返回一个临时的 `reference` 类型的对象——没有别的了。所以, 赋值语句:

```

int x = f[i];

```

等价于:

```
int n = f.operator[](i);
```

变成:

```
int n = reference (f, i);
```

程序清单 10.12 用运算符[]来区分左值和右值

```
// brackets.cpp
#include <iostream>
using namespace std;

class Foo
{
public:
    class reference
    {
        friend class Foo;

        Foo& foo;
        int index;
        reference(Foo& f, int idx) : foo(f), index(idx)
        {}

    public:
        int operator=(int val)
        {
            cout << "assigning " << val << " to position "
                  << index << endl;

            //通常, 在此处理左值...

            return val;
        }
        operator int() const
        {
            cout << "assigning from position " << index << endl;

            //通常, 在此处理右值...

            return 1;
        }
    };

    reference operator[](int idx)
    {
        return reference(*this, idx);
    }
};
```

```

    }
};

main()
{
    Foo f;
    int x = f[0];
    cout << "x == " << x << endl;
    f[0] = 1;
    return 0;
}

```

//输出:

assigning from position 0

x == 1

assigning 1 to position 0

编译器自然要寻找从 `reference` 到 `int` 的类型转换, 在 `Foo::reference::operator int` 可以找到这种类型转换, 它在这里这个程序中返回 1, 但是可以用右边的内容来代替 `Foo` 的右值。

因为在左值的情况下, 语句:

```
f[i] = 1;
```

变成:

```
reference(f, i) = 1;
```

这在左值的情况下调用:

```
reference(f, i).operator=(1);
```

10.10 新风格强制类型转换

强制类型转换是告诉编译器允许你做有一些潜在危险的事情, 因为你知道自己正在做什么。虽然, 太多的情况下自己不知道自己正在做什么 (这里没有冒犯的意思!), 并且花费大量的时间跟踪危险的指针类型转换错误或者类似的错误。有趣的是程序员为了许多不同的目的使用单独的强制类型转换符号:

- 重新解释一个对象的位模式;
- 使一个值变宽或变窄;
- 越过一个类层次;
- 去掉 `const` 或者 `volatile`;
- 做一些实现依赖的魔法。

C++中新风格的强制类型试图通过区分常用的不同强制类型转换操作来减少错误的发生。如果进行了不正确的强制类型转换, 这些新型的强制类型转换将会在编译期或运行期报错, 这根据具体情况而定。

新风格的强制类型转换机制是:

```
dynamic_cast
```

```
static_cast
reinterpret_cast
const_cast
```

dynamic_cast 操作符提供了安全的向下强制类型转换。可以经常把一个指向派生类的指针赋值给指向基类的指针，例如：

```
class B{...} ;
class D : public B {...} ;
D d;
B* bp = &d;
```

这是安全的，因为 D 和 B 服从“is-a”关系。如果被引用的对象实际上是 D，则从指向 B 的指针给指向 D 的指针赋值才有意义。这种指针转换的类型叫做向下强制类型转换，因为它沿着类的层次向下（大多数人把基类看成是在类语法中的最高类）。为了决定这样的一种强制类型转换是否安全，需要有运行期类型信息（RTTI）。C++ 的编译器为所有的内置和多态类型的对象保持运行期类型信息（至少有一个虚函数类型）。如果向下强制类型转换不安全，则 **dynamic_cast** 操作符返回一个空指针，否则它返回强制转换的原始指针给新类型，如下所示：

```
D*dp = dynamic_cast<B*> bp;
if (dp)
{
    //允许对全体使用 dp
    //指定一个 D 对象
}
```

另外三个新风格的强制类型转换操作符是编译期机制的。所有新风格的强制类型转换都用模板形式的三角括号把目标类型括起来。

当确实知道指向 B 的指针指向了指向 D 的指针时可使用 **static_cast**，因为运行期的检查变得没有必要了。也可用它来显式地调用标准的类型转换和用户自定义的类型转换，这可能使 **static_cast** 成为从 C 移植过来的代码中应用最广泛的强制类型转换种类。

可用 **reinterpret_cast** 做除了导航类层次或去掉 **const** 和 **volatile** 外的任何事情。如果需要来回地对整数和指针类型进行强制类型转换，包括从 0 转换成空指针，可以使用这种强制类型转换。它和传统的强制类型转换符号一样不安全，除非不愿大量地使用这种转换，因为存在着其他三种新风格类型转换，也因为像它像一个剧痛的手指伸了出来：

```
int j;
int *ip = &j;
char *cp = reinterpret_cast<char *>(ip);    //相信我！
正如所猜测的那样，const_cast 使得想去掉 const 的意图变得十分清楚：
void B::f() const
{
    //改变一个常量对象的数据成员
    const_cast<B*>(this)->state = ...;
    ...
}
```

除了 **const** 和（或者）**volatile** 限定符（“cv_qualifiers”）的存在，实参的类型必须与括号

中的目标类型相同。其他新风格的强制类型转换都不能改变 `cv_qualifiers` 的状态。

10.11 小结

- C 和 C++ 提供隐式类型转换以简化混合模式运算。
- 在运算中整数类型可以根据需要而升级。
- 浮点运算由标准的类型转换来“平衡”。
- 函数原型帮助编译器提供隐式类型转换。
- 有时候不希望用户定义类型的隐式类型转换。
- 当想避免隐式类型转换时，可以显式地声明带有单个参数的构造函数。
- 强制类型转换使得类型转换变得清楚。
- 尽量使用函数风格的强制类型转换，而不是 C 风格的强制类型转换（即构造函数语法）。
- 当适用时使用新风格的强制类型转换（即除了使用构造函数调用时，总是使用）。
- 当适用时尽量用 `const`。
- 首选语义常量而不是位逻辑常量。
- 可变的存储类使得在大多数情况下去掉 `const` 变得没有必要。
- RTTI 允许安全的向下强制类型转换。

练习答案

由于 `x` 的值是 `0x7C` (124)，在表达式：

`x + x`

两个 `x` 的出现被升级为 `0x0000007C`，相加为 `0x000000F8`。没有发生更多的类型转换。`y` 值是 `0xF8`，然而因为置高位的原因，升级后结果是 `0xFFFFFFFFF8`。

可见性

11.1 名字中包含什么

在 C 或者 C++ 程序中每个以字符或者下划线开头的标记，除了关键字和宏之外，它命名了下面实体中的一个：

- 数据对象
- 函数
- 类型定义 (typedef)
- 类/结构体、共用体或者枚举类型的标签
- 类/结构体和共用体的成员
- 枚举常量
- 标志
- 模板

这些实体在程序中的一定时间和一定地点起作用，这取决于这些声明如何出现以及出现在何处。不管你是否意识到了这点，当你声明标识符时就确定了它的作用域、生命期、连接和名字空间。在本章中我将像标准 C++ 所定义的那样解释这些相关的概念。

11.2 作用域

一个标识符的作用域是程序文本的区域，在这个区域中可以使用这个标识符，或者换句话说它是可见的。声明把标识符引入在编译期确定的作用域中。在标准 C++ 中有 5 种生存期类型：

局部——在一对匹配的括号{}中的邻近区域，这个区域开始于说明符第一次出现的地方并结束于第一个随后的结束括号。

函数原型——从标识符在函数原型出现的地方起到函数原型结束的地方为止的区域。

函数——函数的整个函数体（只有标志有函数作用域）。

类——一个包括类的声明，也包括所有的成员函数体、默认参数和类构造函数的初始化列表的区域。

名字空间——一个标识符第一次在命名的和未命名的名字空间实体中出现的区域；全局作用域扩展到翻译单元的结尾，它是名字空间作用域的特例。

假如形参是在它们所在函数最外层的模块中声明，那么它的作用域和函数一样。标识符名字在函数原型中是可选的，如果它们出现的话只能用于文件中。

在程序清单 11.1 中，可选的标识符 `val` 只用于文件中，并且它们在 `f` 的原型外面是不可见的。在 `f` 的定义中正式的形参 `i` 在开始的括号后是立即可见的。由于每一个块引进了一个新的作用域，那么在最里面的块中被 `j` 初始化的 `i` 暂时地隐藏了外部块中的 `i`。在内部块中 `j` 的值可以对 `i` 进行初始化，这是因为 `j` 的声明在先。如果 `j` 按如下声明：

```
{
    int j = i;    //外部的 i
```

程序清单 11.1 说明局部作用域

```
/* scope1.c: 举例说明局部作用域和原型作用域: */
#include <stdio.h>
```

```
void f(int val);
```

```
main()
```

```
{
    f(1);
}
```

```
void f(int i)
```

```
{
    printf("i == %d\n",i);

    {
        int j = 10;
        int i = j;
        printf("i == %d\n",i);
    }
}
```

```
/* 输出:
```

```
i == 1
i == 10
```



```
*/
```

这样 `j` 的值将是 1。这并不存在冲突，因为内部的 `i` 直到下一个语句才可见。一旦被声明以后，标识符就是可见的了。这意味着下面的声明是不正确的形式（如果不是十分无知的话）：

```
{
    int i = i;
```

由于 `int i` 可以完全声明一个名为 `i` 的整数，左边的 `i` 和右边的 `i` 相同，并且 `i` 没有被初始化。

只有标志有函数作用域。函数作用域和一个函数最外部模块的作用域的区别就是标志在整个最外部的模块中都是可见的，甚至在它们被“声明”之前，但是带有模块作用域的标识符只有在它们的声明点后才是可见的。

在 C 中，在任何块或函数形参列表外部被声明的标识符具有文件作用域。这种标识符有时被指定是全局的并从它们的声明点开始到翻译单元结束末尾都是可见的，除非被具有相同名字的块作用域标识符隐藏。在程序清单 11.2 中的 C 程序说明了函数和文件作用域。具有文件作用域的标识符 `i`（被初始化成 13）只有在它的声明后才是可见的，因此试图在 `main` 程序中使用它是错误的。全局的 `i` 在 `f1` 中的任何地方都是无效的，这是因为 `f1` 有自己的名字为 `i` 的形参。`f1` 最里面的块用它自己的 `i` 隐藏了那个形参（它们的类型不必相同）。由于 `f2` 没有声明名字为 `i` 的标识符，因此它访问全局的 `i`。在 `main` 中 `f1` 和 `f2` 的声明只在 `main` 函数中注入这些名字。例如，从 `f1` 中调用 `f2` 就是一个错误。

程序清单 11.2 说明函数和文件作用域

```
/* scope2.c: 举例说明函数和文件作用域 */
#include <stdio.h>
```

```
main()
{
    void f1(int);
    void f2(void);

    f1(23);
    f2();
    return 0;
}

int i = 13;

void f1(int i)
{
    for (;;)
    {
        float i = 33.0;

        printf("%f\n", i);
```

```

        goto exit;
    }
    exit:
        printf("%d\n", i);
    }

    void f2(void)
    {
        printf("%d\n", i);
    }

    /* 输出:
    33.000000
    23
    13
    */

```

11.3 最小的作用域

经过深思熟虑的声明替换可以大大提高一个程序的可读性。大多数编程者还遵循这样一种传统——把所有的声明放在源代码的一个单独部分中, 在一些语言如 Fortran 和 Cobol 中要求这种方式。这样做的优点是总是可以知道到哪里去找变量的定义。但是当程序很大的时候, 这样做将用大量的时间去在标识符声明及其使用的地方之间上下浏览。在微软的 Windows 下编程的人知道匈牙利符号, 这种把标识符类型的编码加到名字中的习惯逐渐成为一种弥补名字声明和应用间距离的方法, 它从以前的 ANSIC 中函数原型和类型检查的不足发展而来。当与模块化设计相联系的时候, 我宁愿替换为 C++ 程序员所知的简单技术, 即在大多数情况下给出一些不必要的长相奇怪的匈牙利名字。我把它叫做最小的作用域, C++ 的设计者 Bjarne Stroustrup 一开始就推荐这种技术。简单地讲, 就是标识符的声明尽可能地靠近第一次使用它的地方。

例如, 从下面的程序片段中能推断出什么?

```

Void f()
{
    int i, j;
    ...
}

```

即使可能只在函数 f 中的一小部分用到 i 和 j, 但是声明表示了它们在函数中的任何地方都是有效的。所以 i 和 j 的作用域可能比它们应该有的作用域要大。例如, 如果 i 和 j 只应用于一定的条件, 可以通过在一个合适的内部模块中声明来限制它们的作用域:

```

void f()
{
    ...
}

```

```

    if (<some condition>)
    {
        int i = 7;
        int j = 2 * i;
        //只在这里用到了 i 和 j
    }
    ...
}

```

同时注意到我直到可以初始化它们的时候才声明它们。C++通过允许声明出现在语句可以出现的任何地方来支持最小的作用域，例如

```

for ( int i = 0; i < n; ++i)
    ...

```

索引 *i* 从它的声明点开始直到循环体的结束都是可见的。最小的作用域有助于提高可读性，因为直到需要它们的时候——给程序增加意义的时候，才看到标识符。

11.4 类的作用域

因为成员名字对它们的类来说是局部的，每一个类创建一个新的作用域。一个类的作用域包括它的声明、所有成员函数实体和构造函数初始化。可以通过下面的方法中的一种来访问类的成员：

1. 作为 `.` 操作符的目标（例如 `x.foo`，这里 `x` 是一个类或者派生类的对象）。
2. 作为 `->` 操作符的目标（例如 `xp->foo`，这里 `xp` 指向类的对象或者派生类的对象）。
3. 作为域作用符的目标和它的类名或基类的名字一起使用（例如 `ios::binary`，它只适用于静态成员）。
4. 来自于类或者派生类的成员函数实体内。

这些规则本身服从于访问规则，它由访问说明符 `public`、`protected` 和 `private` 提示。私有类型成员只有类的友元和类的成员函数可以访问。保护类型成员只有派生类的成员和友元可以访问。公有类型成员可以被全体访问。

程序清单 11.3 有与 `Date` 类相似的定义，程序清单 11.4 包含了实现，程序清单 11.5 是测试程序。如果想知道一个指定日期的月份，不要试图直接访问数据成员，例如，

```

Date d;           //今天的日期
cout<< d.month_;  //访问被拒绝
应该用合适的成员函数来代替它：
Date d;           //今天的日期
cout<< d.month(); //OK

```

程序清单 11.3 `Date` 类的定义

```

// date.h
#ifndef DATE_H
#define DATE_H

```

```
#include <string>
#include <cassert>
using namespace std;

class Date
{
public:
    Date();
    Date(int y, int m, int d);

    //存取器:
    void year(int);
    void month(int);
    void day(int);

    //增变器
    int year() const;
    int month() const;
    int day() const;

    //其他:
    int compare(const Date&);
    operator string() const;
    static bool isleap(int);
    static int endOfMonth(int, int);

private:
    int year_;
    int month_;
    int day_;

    static int dtab[2][13];
};

inline Date::Date(int y, int m, int d)
{
    assert(1 <= m && m <= 12);
    assert(1 <= d && d <= endOfMonth(y,m));
    year_ = y;
    month_ = m;
    day_ = d;
}

inline int Date::year() const
{

```

```
    return year_;
}

inline int Date::month() const
{
    return month_;
}

inline int Date::day() const
{
    return day_;
}

inline void Date::year(int y)
{
    year_ = y;
}

inline void Date::month(int m)
{
    assert(1 <= m && m <= 12);
    month_ = m;
}

inline void Date::day(int d)
{
    day_ = d;
}

inline bool Date::isleap(int y)
{
    return y%4 == 0 && y%100 != 0 || y%400 == 0;
}

inline int Date::compare(const Date& r)
{
    int ydiff = year_ - r.year_;
    int mdiff = month_ - r.month_;
    return ydiff ? ydiff
        : (mdiff ? mdiff
            : day_ - r.day_);
}

inline int Date::endOfMonth(int y, int m)
{
    assert(1 <= m && m <= 12);
    return dtab[int(isleap(y))][m];
}
```

```

    }

```

```

#endif

```

`Date` 类的大多数成员都是非静态的。每一个日期对象都有自己的对每一个非静态数据成员（即 `month_`、`day_` 和 `year_`）的拷贝，并且每一个非静态成员函数必须结合 `Date` 类对象来调用（例如 `d.month`）。静态成员属于整个类而不是单个的对象。（一些面向对象的语言把静态数据成员叫做类变量，把静态成员函数叫做类方法，而把非静态数据成员和非静态成员函数分别叫做实例变量和实例方法）。把域作用符和类名联系起来使用可以允许直接访问类的静态成员。既然 `isleap` 是公有成员函数，那么可以用它来确定任何一年是否是“闰年”，如下所示：

```

int y = 1994;
if (Date::isleap(y) )
    //是闰年.....
else
    //不是闰年.....

```

必须在文件作用域中明确定义静态数据成员（见程序清单 11.4 中 `dtab` 的定义）。

如在程序清单 11.3 的 `Date::day(int)` 所示，其他类成员函数在成员函数实体内的作用域中，因此可以不用显式的域作用符直接访问它们。每一个非静态成员函数都有一个隐藏的形参，它指向所调用对象，并且可以通过使用关键字 `this` 来得到。在 `Date::day(int)` 内，编译器解释 `day_`、`isleap` 和 `dtab` 的用法就好像已经写了 `this->day_`、`Date::dtab` 和 `Date::isleap` 一样。静态成员没有 `this` 指针，这是因为没有相联系的对象。因此在静态成员函数如 `isleap` 中不加修饰地用标识符 `day_` 是没有意义的（它属于什么对象？），并且如果存在相同名字的标识符，那么就会从封装的作用域中查询它。静态成员函数只能直接查阅枚举、嵌套类的名字和其他的静态成员。

程序清单 11.4 `Date` 类的实现

```

// date.cpp
#include <iostream>
#include <cassert>
#include <ctime>
#include <cstdio>
#include "date.h"
using namespace std;

// 必须在类定义外面初始化静态成员：
int Date::dtab[2][13] =
{
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};

Date::Date()
{

```

```

    // 获得今天的日期
    time_t tval = ::time(0);
    struct tm *tmp = ::localtime(&tval);

    month_ = tmp->tm_mon+1;
    day_ = tmp->tm_mday;
    year_ = tmp->tm_year + 1900;
}

Date::operator string() const
{
    char buf[9];
    sprintf(buf, "%04d%02d%02d", year_, month_, day_);
    return string(buf);
}

```

程序清单 11.5 测试 Date 类

```

// tdate.cpp
#include <iostream>
#include "date.h"
using namespace std;

main()
{
    Date d1, d2(1951, 10, 1);
    cout << "d1 == " << d1 << endl;
    cout << "d2 == " << d2 << endl;
    cout << "d1.compare(d2): " << d1.compare(d2) << endl;
    cout << "d2.compare(d1): " << d2.compare(d1) << endl;
}

// 输出:
d1 == 19970104
d2 == 19511001
d1.compare(d2): 46
d2.compare(d1): -46

```

既然 `Date::day_ (int)` 的代码是作用域的一部分，那么任何局部的标识符或者与全局标识符有相同名字的成员标识符在封装的作用域中隐藏了和它们名字相同的标识符。例如，假设已经定义了一个名字为 `day_` 的标识符（这种情况不可能发生），在所有非静态日期成员函数中它将被数据成员 `day_` 隐藏。然而可以通过使用不带前缀的域作用符来访问一个全局标识符：

```

day_ = ::day_;           // 用全局的 day_ 给成员赋值

```

实际上, C++ 允许“不隐藏”全局标识符。这有另一个例子:

```
int i = 10;
main()
{
    int i = 20;
    cout << "local i: " << i << endl;
    cout << "global i: " << ::i << endl;
    return 0;
}
```

//输出:

```
local i: 20
global i: 10
```

可以通过用名字空间的名作为作用域标识符而“不隐藏”任何出现在名字空间中的标识符(见“名字空间作用域”部分)。使用目标文本的域作用符显式地来限定全局函数是一个好的做法,这样可以避免成员名字的冲突(见程序清单 11.3 中构造函数 `Date::Date` 的实现)。

当要把一个名字约束成为独一无二的声明时, C++ 编译器首先在封装的作用域中查找那个名字。一旦找到,编译器便进行访问权限检查并应用重载作用符。例如,考虑下面的这些类:

```
class B
{
public:
    void f();
}
class D : public B
{
    void f();           //私有的!
};

main()
{
    D d;
    d.f();
}
```

在 D 类的作用域中有一个名字为 f 的函数,因此编译器停止查找。由于 `D::f` 是私有的,编译器诊断出错误。暂且假设 `D::f` 的标记是 `void f(int)`。这样仍然无法把表达式 `d.f` 约束为 `B::f`,因为在 D 类中的名字隐藏在 B 类中的。换句话说,派生类的作用域被基类的作用域所覆盖。如果想用 D 类的对象执行 `B::f`,必须显式地写为:

```
d.B::f();
```

11.5 嵌套类

除了数据成员、成员函数和枚举常量外,还可以在类定义中定义其他的类。例如,一个

叫做 **copy-on-write** 的常用字符串类实现技术允许几个字符串对象指向相同的文本（见程序清单 11.6）。当其中一个字符串改变了它的内容的时候它只是做独立的拷贝（当字符串被用做形参传递时这样做可以节省时间——局部只读处理用一个“共享”的拷贝）。一个单独的类（**Srep**）处理涉及下面文本的连接和取消连接的具体细节。没有理由使 **Srep** 类是全局可访问的，因此我在 **String** 类中定义了它。在程序清单 11.7 中的实现说明了如何使用域作用符去访问 **Srep** 类的成员。例如，为了通知编译器我想定义构造函数 **Srep**，我用到了标识符 **String::Srep::Srep**。

程序清单 11.6 一个 copy-on-write 字符串类的设计

// str.h: 有参考记数的字符串类

```
class String
{
public:

    // 构造函数/析构函数
    String(const char *s);
    String(const String& s);
    ~String();
    // 省略其他的成员 .....

private:
    class Srep
    {
    public:
        Srep(const char*);
        ~Srep();
        char *rep;
        size_t count;
    };

    Srep *rep;
};
```

在平常情况下一个嵌套类中的名字隐藏了封装类中的名字，可以用域作用符访问外部实例（当然，只有在它是公有的时候才能这样做，通用的访问规则来回地应用于嵌套类）。例如，在下面的情况下：

```
class A
{
    static int i;
    class B
    {
        int i;
        void f() { cout << i + A::i; }
    };
};
```

```
};
```

仍然可以用表达式 `A::i` 在 `A` 类中访问 `i`。

程序清单 11.7 程序清单 11.6 的实现

```
// str.cpp
#include <iostream.h>
#include <cstring>
#include "str.h"
using namespace std;

String::String(const char *buf)
{
    if (!buf)
        rep = new Srep("");
    else
        rep = new Srep(buf);
}

String::String(const String& s)
{
    rep = s.rep;
    rep->countl++;
}

String::~String()
{
    if (--rep->count <= 0)
        delete rep;
}

String::Srep::Srep(const char* s)
{
    strcpy(rep = new char[strlen(s)+1], s);
    count = 1;
}

String::Srep::~Srep()
{
    delete [] rep;
}
```

11.6 局部类

在一个函数定义中定义的类叫做局部类（见程序清单 11.8）。像其他在函数定义中声明的标识符一样，它的名字局限于函数体中。必须在类定义中定义所有的成员函数。不能在文件

作用域中定义它们，因为类在那里是不可见的。这条规则同样不包括 Local 类的静态数据成员，因为必须在文件作用域里定义它们。不能在函数 f 中别的地方定义函数体，因为函数定义在 C 和 C++ 中不能嵌套。同样的原因，一个 Local 类的成员函数不能引用函数 f 的自动变量。通常成员访问规则同样适用，因此 k 在 Local 类外部的函数 f 中是不可见的。

程序清单 11.8 定义一个 Local 类

```
// local.cpp
#include <iostream>
using namespace std;

int i = 10;

main()
{
    void f();
    f();
}

void f()
{
    static int j = 20;

    class Local
    {
        int k;

    public:
        Local(int i) : k(i) {}
        void a() {cout << k+i << endl;}
        void b() {cout << k+j << endl;}
    };

    Local l(30);
    l.a();
    l.b();
}

// 输出:
40
50
```

11.7 典型的名字空间

标识符能在程序中扮演各种角色。例如在下面这个摘录中，pair 不但是个函数名，还是

一个结构体标志。

```
struct pair {int x; int y};
void pair(struct pair p)
{
    printf("(%d,%d)\n",p.x,p.y);
}
```

编译器保持用在不同角色中的标识符的独立列表，所以不会出现模棱两可的情况。这些列表被称做名字空间。在标准 C 中有四种不同的名字空间：

1. 标志；
2. 结构体、共用体和枚举的标签；
3. 结构体和共用体成员；
4. 普通标识符（即所有其他的标识符：数据对象、函数、类型和枚举常数）。

由于 C++ 中类型的杰出表现，可以在不用 `struct` 关键字的情况下使用结构体标签（这是真的类名），例如：

```
void f(pair p); //结构体的关键字可任选
```

换句话说，所有的 `struct` 名可充当类型名，就好像用与这个标签相同的名字定义了 `typedef` 一样，如下：

```
typedef struct pair pair;
```

这意味着不能在同一作用域中对于不同的类型，把名字既作为结构体标签又作为 `typedef` 使用。既然 `typedef` 属于常用标识符范畴，C++ 用标签名字空间（上面第 2 条和第 4 条）合并那些名字空间。为了兼容性，仍可以用与一个标签相同的名字定义函数，这主要是因为下列 C 的常用作法：

```
struct stat{...};
extern int stat();
```

尽管如此，当把同一个名字用于 `struct` 标签和普通标识符时，问题就来了。在程序清单 11.9 的程序中，局部整型 `pair` 隐藏了全局标志符 `struct`，使得编译器认为这一行是错误的。

```
pair p={pair,pair};
```

可以通过显式地应用关键字 `struct` 或 `class` 来解决这个问题。

```
struct pair p={pair,pair};
```

程序清单 11.9 说明标签和变量共享相同的名字空间

```
// pair.cpp:
#include <iostream>
using namespace std;

struct pair {int x; int y;};

main()
{
    int pair = 0;
    pair p = {pair,pair};    // 错误
```

```

    cout << pair << endl;
    cout << p.x << ',' << p.y << endl;
}

```

11.8 名字空间的作用域

在 C++ 中，文件作用域正好是名字空间的一个特殊情况。一个 C++ 名字空间是个任意命名的声明区域，允许用户按自己的目的将声明集合在一起。

```

Namespace MyNamespace

```

```

{
    void f();
}

```

既然 `f` 属于 `MyNamespace`，它在别的其他的任何地方都默认为不可见的。应用 `f` 时，有三种选择：

1. 用一个完全合格的名字，如

```

# include "mystuff.h" //定义 MyNamespace

```

```

...

```

```

MyNamespace::f();

```

2. 告诉编译器 `f` 任何的使用都与 `MyNamespace::f` 相关，如下：

```

# include "mystuff.h" //定义 MyNamespace

```

```

using MyNamespace::f; //一个 using 声明

```

```

//引入 f 的所有重写

```

```

...

```

```

f(); //调用 MyNamespace::f

```

3. 指示编译器在 `MyNamespace` 中查找它想要发现的任何名字：

```

# include "mystuff.h" //定义 MyNamespace

```

```

using namespace MyNamespace; //一个 using 指示

```

```

...

```

```

f(); //如果没有其他相冲突的 f，调用 MyNamespace::f

```

像上面第二种选择一样的 `using` 声明将选定的名字注入到当前区域中，这样就可将该区域内其他有相同名字的声明覆盖了。一个 `using` 指示为名字解析打开了整个被选定的名字空间。如果在作用域内有相同名字的多声明，编译器将发出通常的诊断。

名字空间原先设想用来解决库使用中的困难。在名字空间出现前，程序供应商不得不在他们所提供的库中把前缀附加到名字中，以便当用户采用其他供应商提供的程序来与这些功能程序结合使用时可以避免名字的冲突。现在可以挑选想要的特征而不用担心编译的不方便和链接错误：

```

# include "vendorA.h" //包括 f() 和 g()

```

```

# include "vendorB.h" //包括 f() 和 g()

```

```

using vendorA::f;

```

```

using vendorB::f;

```

名字空间是唯一的，因为它们声明的区域可以跨越多个文件。例如，可以在两个文件中声明和实现 `MyNamespace`，如下：

文件 1

```
//myname.h
namespace MyNamespace
{
void f();
} //注意：没有分号！
```

文件 2

```
//myname.cpp
void MyNamespace::f()
{
    //实现在这里执行...
}
```

如果不喜欢一个名字空间的名字太长，可以根据自己的喜好声明一个别名：

```
namespace notSoLong=ReallyReallyLongNamespaceName;
using notSoLong::f;
using notSoLong::g;
```

名字空间里的声明不一定非要在同一个翻译单元中连续地出现。属于名字空间的名字集合是名字的共用体，这些名字在所有翻译单元中有相似名字的名字空间模块中声明的名字的共同体。

全局名字空间是在所有名字空间模块之外的声明区域。它就好像用独一无二的名字声明一样，例如

```
namespace unique
{
    ...
}
```

同样的名字在所有翻译单元中应用，因此有连接的标识符能被解决（见下面的“连接”）。如果想使全局标识符对于一个翻译单元是私有的，那么可以用关键字（一个 `la C`）`static` 来限制它们，如下：

```
static int private_integer;
static void private_function();
或者可以用未命名的名字空间来声明它们，该名字空间对任何翻译单元都是独一无二的：
namespace
{
    int private_integer;
    void private_function();
}
```

后者是 C++ 中首选的方法。

11.9 生存期

对象的生命期或存储持续时间是对象被创造到它被销毁之间的持续时间。有静态持续

时间的对象只能在它们首次使用之前被创建并且初始化一次，在程序正常终止时被撤销。有全局作用域的对象和在模块作用域用静态说明符声明的对象，都有静态持续时间。程序清单 11.10 给出了后者的例子。变量 `n` 在程序开始被初始化一次，而且在整个程序中保存它最近被赋予的值。（然而它的作用域只是函数 `count` 的实体。）

在没有 `extern` 或 `static` 说明符的模块内声明的函数参数和对象都有自动的持续时间。这种对象在执行每次进入它们模块的操作时被重建。在每次执行正常地进入模块之后，可能指定的任何初始化也被执行。跳过要求初始化的声明进入任何模块在 C++ 中都是一种错误。当执行失败、或跳转超过了模块的结尾、或从一个函数返回时，在那个作用域内的所有自动变量都被撤销。

程序清单 11.11 的程序通过熟悉的阶乘函数说明了静态持续时间和自动持续时间。标记 `n!`，读“`n` 阶乘”（不用念出来），表示所有正整数从 `n` 乘到 1 的结果。例如：

```
3!=3×2×1
4!=4×3×2×1
(等等)
```

大多数数学课本给出下列相等的递归定义：

```
n!={if n<=1 then1, otherwise n*(n-1)!}
```

程序清单 11.10 说明静态储存

```
// lifetime.cpp: 举例说明静态存储的持续时间
#include <iostream>
using namespace std;

main()
{
    int count(void);
    int i;

    for (i = 0; i < 5; ++i)
        cout << count() << endl;
}

int count(void)
{
    static int n = 0;

    return ++n;
}

// 输出:
1
2
3
4
```

5

在 C++ 中你可以用下面的递归函数简明地表示上面地公式：

```
long fac(long n)
{
    return (n<=1)?1:n*fac(n-1);
}
```

当 n 大于 1 时，`fac` 递归地调用它自身，其参数是在它开始计算时所用数的基础上减去 1 后的数。操作临时地中止当前的作用域，并用 n 的自身拷贝创建一个新的作用域。这样一直持续到最深层嵌套 n 的拷贝等于 1。作用域终止并将 1 返回给调用它的作用域，如此向上回到原始的调用。例如，考虑表达式 `fac(3)` 的执行：

```
fac(3): return (3<=1)?1:3*fac(2);
```

程序清单 11.11 举例说明递归和静态存储

```
/* recurse.c: 举例说明递归和存储的持续时间 */
#include <stdio.h>
```

```
main()
{
    long n;
    long fac(long);

    fputs("Enter a small integer: ",stderr);
    scanf("%ld%c",&n);
    printf("\n%ld! = %ld\n",n,fac(n));
    return 0;
}
```

```
long fac(long n)
{
    static int depth = 0;
    auto long result;
    void print_current(int,long);

    print_current(++depth,n);
    result = (n <= 1) ? 1 : n * fac(n-1);
    print_current(depth--,result);

    return result;
}
```

```
void print_current(int depth, long n)
{
    int i;

    // 缩进以表示深度
```



```

    for (i = 0; i < depth; ++i)
        fputs("  ", stdout);

    printf("%ld\n", n);
}

```

/* 输出:

Enter a small integer: 3

3

2

1

1

2

6

3! = 6

*/

这次调用 fac (2):

fac(2):return (2<=1)?1:2*fac(1);

依次调用 fac (1):

fac(1):return (1<=1)?1:fac(0);

返回值 1:

fac(1); return 1;

fac (2) 重新开始并把下面的内容返回给 fac (3):

fac(2):return 2*1;

它把 6 返回给原始的调用。

程序清单 11.11 中的程序通过用语句绕接阶乘公式来追踪阶乘计算以打印输入函数的值和所算得的输出值。连同静态变量 `depth`, 这个程序跟踪了递归已经嵌套的深度。既然 `depth` 有静态持续时间, 它在第一次调用 `fac` 之前被分配和初始化一次, 并跨过函数调用来保存其值 (包括递归函数调用)。只有自动变量如 `fac` 函数里的 `n`, 在每次递归调用中被复制。关键字 `auto` 是纯文本的, 因为所有模块作用域变量在默认时是自动的。

存储在自由存储区内的对象由 `new` 操作符返回, 并有动态存储持续时间, 它非常有意义以至于值得用单独的一章来讨论 (参见第 20 章)。

11.10 临时对象的生存期

当在运行期计算表达式时, 编译器有时产生临时对象。例如, 表达式 `x=a+b+c` 可能需要一个临时对象。也许像下面这样的语句将被执行:

```

temp=b+c;
x=a+temp;
//这里撤销 temp

```

为了保护资源，对编译器来说尽早地销毁临时对象是很重要的，但又不能过早。考虑下面类的声明：

```
class string
{
public:
    string(const char*);
    operator const char*();
    string operator+(const string&); //连接
    //其他省略
};
```

const char* 操作符允许把字符串参数与标准 C 库函数一起使用，例如，

//好格式的例子

```
string s,t;
```

...

```
printf("%s\n", (const char*)(s+t));
```

当然希望表达式 `s+t` 中的临时变量能坚持足够长的时间以便 `printf` 处理它。如果试着将它分成两个语句中，结果会令人失望的。

//错误格式的例子

```
string s,t;
```

...

```
const char *p=s+t;
```

```
printf("%s\n",p);
```

在标准 C++ 中，被赋值为 `p` 的临时变量不能坚持到 `printf` 语句，如上面错误格式的代码。规则是临时变量可持续到它所在的整个表达式的结尾。在大多数情况下，这就意味着直到封装语句的结尾，因此临时变量不能从一个语句持续到下一个语句。一个完整的表达式不是其他表达式的一部分，如：初始化语句；`if`、`while`、或 `switch` 的控制表达式；`for` 语句中的表达式；从一个函数返回的表达式。

规则的唯一例外的是绑定临时变量的引用。例如，考虑下列语句：

```
class T{/*无论...*/};
```

```
f();
```

```
Const T &r=f();
```

正如所希望的那样，`f` 返回的临时对象在生存期内持续时间和 `r` 一样长。

11.11 连接

根据连接规则，在一个作用域的标识符可以引用在另一个作用域中与它有相同名字的对象。任何在一个合法的 C++ 程序中声明的名字都将是下列连接属性中的一种。

外部连接——在一个程序中跨过翻译单元连接名字。

内部连接——贯穿一个单独的翻译单元连接名字。

无连接——某些名字是独一无二的，因此它们没有连接。

没有用关键字 `static` 声明的函数和其他全局标识符都有外部连接。每种这样的对象必须只有一个定义，但也可能有关于那个定义的许多声明。如果下列的声明在一个文件的全局作用域出现：

```
//file1.cpp
```

```
int x;
```

那么它就可以被用在另一个文件的任何作用域中，在此作用域中有下面的声明：

```
//file2.cpp
```

```
extern int x;
```

`extern` 说明符实质上说明：“在某一翻译单元中找一个名为 `x` 的全局对象”，这个对象可以是与另一个翻译单元中有外部连接的对象，或是与同一个翻译单元有内部连接的对象。当声明（不是定义）一个函数时，`extern` 限定词暗含：

```
//file1.c
```

```
int f(void)
```

```
{
```

```
    return 1;
```

```
}
```

```
//file2.cpp
```

```
int f(void); //函数的 extern 说明符到 file1.cpp 文件中 f 的隐式连接
```

在名字空间中（包括全局名字空间）用 `static` 说明符声明的函数和对象具有内部连接。在没有命名的名字空间中的声明也有内部连接。内部连接的标志符只在其翻译单元中可见。关键字 `static` 的使用对前面讨论的静态存储持续时间说明符几乎不起作用。记住用下列的假公式是一个好主意：

`static+模块作用域==静态存储持续时间`

`static+全局作用域==内部连接`

`static` 的第一个使用影响生命期，第二个则对连接产生影响。

某些总是独一无二的程序实体没有连接。它们包括具有无 `extern` 说明符模块作用域的对象、函数参数以及函数或对象之处的其他任何东西，如标志、标签名、成员名、`typedef` 名和枚举常量。

程序清单 11.12 和 11.13 的源文件构成了一个单独的可执行程序，它说明了不同类型的连接。在程序清单 11.12 中文件作用域中的整数 `i` 有外部连接，因为它没有 `static` 说明符。如果是用 `extern` 说明符声明的变量则可以用另一个文件中与它同名的变量来引用，如程序清单 11.13 所示。（外部连接时，同一个对象拥有两个定义是错误的，例如两个 `i` 的定义既不用 `static` 也不用 `extern` 来限定。）

程序清单 11.12 举例说明外部连接

```
// linkage1.cpp: 与 linkage2.cpp 相链接
```

```
#include <iostream>
```

```
using namespace std;
```

```
namespace
```

```
{
    void f1(int);           // 内部的
    void f2(void);          // 内部的
    int k = 7;              // 内部的
}

main()
{
    extern void f1(int);     // 内部的
    extern void f2(void);    // 内部的
    extern void f3(void);    // 外部的

    f1(23);
    f2();
    f3();
    return 0;
}

int i = 13;                // 外部的

void f1(int i)              // 内部的
{
    for (;;)
    {
        float i = 33.0;     // 无链接
        cout << i << endl;

        extern int k;        // 外部的!!!
        cout << k << endl;
        goto exit;
    }

exit:                        // 无链接
    cout << i << endl;
}

void f2(void)               // 内部链接
{
    cout << i << endl;
}

// 输出:
33
8
23
13
16
```

程序清单 11.13 和程序清单 11.12 相连接

```
// linkage2.cpp: 与 linkage1.cpp 链接
#include <iostream>
using namespace std;

extern int i;           // 外部的
int k = 8;             //外部的

namespace
{
    int j = 3;         // 内部的 (未命名的名字空间)
}

void f3(void)          //外部的
{
    cout << i+j << endl;
}
```

函数 f1 和 f2 有内部连接, 因为它们用了 static 说明符。在 f1 中命名为 i 的浮点对象没有连接, 因为它在模块作用域中声明并没有 extern 说明符。程序清单 11.13 中的整数 j 有外部连接, 因为它在没有命名的名字空间里, 而函数 f3 有外部连接, 因为没有 static 说明符。

程序清单 11.12 的下列三行需要特殊解释:

```
extern void f1(int);    //内部连接
extern void f2(void);   //内部连接
extern void f3(void);   //外部连接
```

既然 extern 说明符意味着“把某些对象和全局作用域相连接”, 在 main 中 f1 和 f2 的声明分别和在同一个文件中与它们有同样名字的函数相连接, 该函数恰巧有内部连接。在 main 中的 extern 引用前面在文件作用域中声明 f1 和 f2 是很重要的, 不然编译器会假定它们有外部连接 (像 f3 那样), 这将与稍后在文件里的实际的函数定义相冲突。

不像 C, const 对象在 C++ 的文件作用域中都有内部连接, 除非有 extern 说明符。这允许把 const 放在包含文件中的习惯做法, 并不会引起连接器报错。如果希望 const 对象有外部连接, 用 extern 限定每一个对象并只初始化它们一次, 如下:

```
File 1
extern const int x=10;
File 2
Extern const int x; //在文件 1 里引用 x
```

一个声明在名字空间作用域类名 (即一个非局部的类, 它意味着用户可能将看见的每个类的名字!) 有外部连接, 因此它的成员函数、静态数据成员、枚举量和嵌套类也有外部连接。静态成员有外部连接, 这就解释了为什么必须显式地在文件作用域中定义它们。它们实质上也是服从作用域作用域和成员访问规则的全局标识符。非内联类成员函数有外部连接, 这允许从类声明中的一个独立的文件定义它们。非成员内联函数也有内部连接。

因为关键词 `inline` 对翻译器来说仅仅是一个提示，因此对内联成员函数的连接进行分类是很困难的。如果一个函数是内联的，可以来想像它有内部连接（无论如何，将它们放到类的头文件中是一个惯例）。如果编译器不能插入内联代码，必须创建一个外联函数。虽然一些实现喜欢只用一个外联函数以节省代码空间，但是这暗含着外部连接。标准中规定了在程序中一个内联函数必须只有一个定义。

11.12 类型安全连接

像第 1 章所解释的那样，在标准 C 中不能保证在一个翻译单元中定义的函数会在另一个翻译单元中正确地使用。例如，考虑源文件：

```
File 1
Void f(double x)
{
    ...
}

File 2
Extern void f(int);
...
f(1);
```

在文件 1 里定义的函数要求一个 `double` 型实参，但有人在文件 2 中给用户错误的信息，程序就此被中断。传统 C 规则要求两个文件都包含 `f` 正确原型的公用头文件。C++ 为程序员承担这个责任并且通过类型安全连接的概念把这一责任转给开发环境，这保证了上面的 2 个 `f` 版本不相连。大多数实现通过将正式参数的信息编码为对连接器可视的隐蔽名字（这通常被称为名字混乱）来完成上述工作。例如，一个编译器可能把文件 1 中的名字编码成：

```
f_Fd // “f 是一个带有双精度参数的函数”
```

而文件 2 中的名字编码为：

```
f_Fi // “f 是一个带有整型参数的函数”
```

由于连接器看见两个不同的名字，则从文件 2 到文件 1 对 `f` 的引用没有被解决，这就导致连接错误。因此，除非正确调用自己的函数，否则不能产生一个可执行程序！

11.13 “语言”连接

C++ 有很多原因吸引 C 程序员。他们所知的大多数内容在 C++ 中没有被改变。在习惯于 C++ 的语法和特性之前，他们可以把 C++ 当作一种“更好的 C”来使用，从而使他们轻松地进入面向对象的编程。但有能力的程序员还需要一些更重要的内容：能够利用现成的 C 代码，甚至不用重新编写。支持重写的名字编码，这看起来不太可能。对函数 `f(double)` 来说，当它在 C 中的名字是 `f`（或更平常的 `_f`），而在 C++ 中的名字是 `f_Fd`，那么怎样从 C++ 程序中

调用 C 函数呢？可以指示编译器通过连接说明来进行与 C 程序的连接：

在 C++ 文件里：

```
extern "C" f();
```

`extern "C"` 说明符告诉编译器根据 C 的规则生成连接名，这样连接器就可以在现有的 C 对象代码中找到 `f()`。一个实现可能支持除 “C” 外的其他连接语言说明符。语言引用是函数类型的一部分，但不是它的签名（用在函数重载域的参数类型的顺序）。连接说明符只能在名字空间的作用域中出现。

11.14 小结

- 一个标识符的作用域是程序文本可见的区域。
- 一个局部作用域是由花括号 {} 限定了的程序文本的一个连续区域。
- 类成员的声明必须在程序文本的一个词法临近的区域中出现。成员函数和静态数据成员的实际定义可以（通常是应该）在另外的翻译单元中。
- 名字空间作用域能够跨越多重翻译单元。一个名字空间由在所有翻译单元中它的所有声明单元组成。
- 在文件作用域内的声明在全局名字空间范围内。在一个完整的程序中只有一个全局名字空间。
- 嵌套作用域中的声明在封装生存期内隐藏了具有相同名字的声明，不考虑类型、签名和访问权限等其他条件。
- 一个编译器通过先考虑的作用域，然后考虑访问说明，最后考虑重载域来解决名字绑定。
- 一个对象的生命期是从它被创建到它被撤销之间的持续时间。有三种类型的储存持续时间：自动、静态和动态。
- 静态对象在它们第一次被使用之前初始化一次。自动对象则在执行每次进入它们的作用域时都被初始化。
- 标志符可以有外部连接（在多重翻译单元内可见）、内部连接（只在单独的翻译单元中可见）和没有连接。
- 有外部连接的对象必须在一个被命名的名字空间中定义（包括被秘密命名的全局名字空间）。
- 在未命名的名字空间内标志符有内部连接。
- 实现可能支持与其他编程语言实体的连接，`extern "C"` 被所有的实现所要求。

控制结构

25 年前 Edsger Dijkstra 写下了题目为“GOTO 语句应被视为是有害的”这一著名的信件。正如这封信件标题所喻示的那样，他强烈地强调了在编程中不要使用分支结构¹。在另一封信里，他写道：“…作为一个程序员，有时我知道不用任何形式的 goto 语句我可以过得很愉快，但是…在此期间经过我考虑后的看法是有 goto 语句我不可能过得很愉快”²。在这之后一个声名狼藉的完全废弃 goto 结构的运动开始了。教授拒绝接受学生那些使用 goto 语句的编程作业。语言被设计为没有 goto 结构的形式。所有这些的积极方面是作为一个行业我们学会“结构化”地思考，即用一种更有逻辑的和更可维护的方式来组织我们的软件。消极方面是有时 goto 正是博士要求使用的。当今几乎所有被广泛应用的编程语言都支持 goto 的一些形式。Java 语言是一个典型的例外，它使用标记中断来代替 goto。

在这一章中我将讨论可用于组织 C/C++ 程序逻辑的机制。这不但包括结构化编程的 goto-less 结构，还包括从简单到复杂的分支技术：break、goto、用 setjmp/longjmp 的非局部跳转和信号。

12.1 结构化编程

1966 年 Bohm 和 Jacpini 用数学方法证明了只用三个结构和任意数量的布尔型标志就能表示任何算法。这三种结构是：

1. 顺序；
2. 选择（例如，if-else, switch）；
3. 循环（例如，while, for, do）。

我们通常把只使用这些机制的程序叫做结构化程序。不幸的是，当结构化设计的发起人开始向计算机产业艰难地推销他们的产品时，大多数专业编程人员正在用 Cobol、Fortran IV

和汇编语言谋生，而 Basic 正在教育环境中流行。在这些语言当中，只有 Cobol 具有 else 结构。如果告诉一个 Fortran 程序员他不应该使用 goto，通常的结果是，他会告诉你 where to go to（你应该到哪儿去）。此外，我们该为我们专家的这种将复杂的逻辑压缩到尽可能短的几行的能力而感到骄傲——如果呆子不能理解这种“优雅”那么谁还会注意它呢？没有 goto 我们不可能完成这项伟大的工程。就结构化编程革命而言，P.J.Plauger 曾经说过，“我们的皈依者在死脑筋的汇编语言程序员鼻子底下灵活地操纵有趣的少量消息，而这些汇编语言程序员一边沿着扭曲的逻辑不断地向前跑，一边说：‘我和你打赌不能使程序构造化。’无论是 Bohm 和 Jacopini 的验证还是我们编写结构化代码的不断成功都不能说服这些人，直到某一天他们将自己说服自己。”

程序清 12.1 Hi-lo 游戏的 Basic 程序

```
100 rem A very old Basic program to play Hi-Lo
110 print "Think of a number between 1 and 100"
120 print "If you don't cheat, I'll figure it out"

130 print "in seven guesses or less!"
140 lo = 1
150 hi = 100
160 if lo > hi then print "You cheated!" : goto 240
170 g = int((lo + hi) / 2)
180 print "Is it";g;" (L/H/Y)?"
190 input r$
200 if r$ = "L" then lo = g+1 : goto 160
210 if r$ = "H" then hi = g-1 : goto 160
220 if r$ <> "Y" then print "What? Try again..." : goto 190
230 print "What fun!"
240 print "Wanna play again?"
250 input r$
260 if r$ = "Y" then 140
```

程序清 12.2 Hi-lo 的 C++ 程序

```
// hi-lo.cpp
#include <iostream>
#include <cctype>
using namespace std;

main()
{
    cout << "Think of a number between 1 and 100.\n"
        << "If you don't cheat, I'll figure it out\n"
        << "in seven guesses or less!"
        << endl;
```

```

bool done = false;    //循环控制变量

while (!done)
{
    char response;
    int guess;

    // 玩游戏:
    bool found = false;    //循环控制变量

    int lo = 1, hi = 100;
    while (!found && lo <= hi)
    {
        // 获得 guess:
        guess = (lo + hi) / 2;
        cout << "Is it " << guess << " (L/H/Y): " << endl;
        cin >> response;
        response = toupper(response);

        // 缩小查找范围:
        if (response == 'L')
            lo = guess + 1;
        else if (response == 'H')
            hi = guess - 1;
        else if (response != 'Y')
            cout << "What? Try again..." << endl;
        else
            found = true;
    }

    //打印结果:
    if (lo > hi)
        cout << "You cheated!" << endl;
    else
        cout << "Your number was " << guess << endl;

    cout << "Wanna play again? " << endl;
    cin >> response;
    done = toupper(response) != 'Y';
}

//执行示例(数字是 37 和 99):
c:>hi-lo
Think of a number between 1 and 100.
If you don't cheat, I'll figure it out

```

```
in seven guesses or less!
```

```
Is it 50 (L/H/Y): h
Is it 25 (L/H/Y): l
Is it 37 (L/H/Y): y
Your number was 37
What fun!
Wanna play again? y
Is it 50 (L/H/Y): l
Is it 75 (L/H/Y): l
Is it 88 (L/H/Y): l
Is it 94 (L/H/Y): l
Is it 97 (L/H/Y): l
Is it 99 (L/H/Y): l
Is it 100 (L/H/Y): h
You cheated!
Wanna play again? n
```

程序清单 12.1 是一个玩“Hi-lo”游戏的经典的 Basic 程序。给出一个 1 到 100 之间的数，然后只要告诉计算机它猜的数比你给的数是大、是小还是正好是你给的数。只要给出诚实的回答，二进制搜索算法将最多用 7 次则可猜到所给的数字（因为 $\text{ceil}(\log_2 10) = 7$ ）。将这个程序与程序清单 12.2 的程序相比较。Basic 程序不到 C++ 程序长度的一半，但它更可读吗？更易维护吗？程序的形状有明显的逻辑结构吗？不，不，没有！

作为 Bohm 和 Jacopini 的工作结论——当设计一个逻辑过程时，把选择和循环作为想法的基本工具使用是很自然的。用结构化的文字或伪代码（一种简明的、由循环和条件所管理的语言描述）很快变得十分流行。对条件的描述与任何支持 if-else 结构的语言对条件的描述是十分相似的。

```
if <条件>
    执行这些程序...
else
    执行程序
```

为了从一组相关的值中进行选择，有 case 语句：

```
case x of
1: 执行这些程序...
2: 执行那些程序...
否则: 执行别的程序...
```

对于各种不同风格的循环有单独的控制手段。大多数情况下需在进入循环体前检验一下它的条件：

```
while<条件>
    执行程序...
for 循环对于必须执行指定次数的过程是很有用的，如
for i=1 to n
```

```

    执行一些由 i 决定的程序
或为了处理事情序列:
for 名册上的每个学生
    进行计算...
少数情况下需要进入循环体后再检验控制条件的循环:
repeat
    输入
until 输出范围在 [1, 5] 内

```

程序清单 12.3 是一个合并两类文本文件过程的伪代码描述。它从每个文件读取一行并且按字典顺序将排在前面的行打印出来。过程重复直到一个文件为空时为止，在此点它正好输出另一个文件的剩余部分。

用一种直接易懂的方法将循环条件联系到设计是很重要的。为确保这一点，可以在循环体开始的地方根据问题的空间插入一条说明循环条件意义的注释。如果它清楚地表达了你的思想，那么就已经完成了一个有效的循环。也可以在循环末尾做同样的事。程序清单 12.5 合并过程的实现就是一个例子。（输入数据在程序清单 12.4 中）

程序清单 12.3 合并两个文本文件过程的伪代码

```

open and read first line from each file
while (neither file has been exhausted)
{
    determine which line comes next
    print that line
    get a fresh line from the corresponding file
}

empty the remaining active file, if any
close files

```

程序清单 12.4 程序清单 12.5 和 12.7 的输入文件

```

FILE1.DAT
brown
fox
quick
the
FILE2.DAT
dog
jumped
lazy
over
the
FILE3.DAT
and
cow

```

```
jumped
moon
over
the
the
```

程序清单 12.5 程序清单 12.3 的 C 实现

// merge1.cpp: 将两个已排序文件合并到标准输出

```
#include <iostream>
#include <fstream>
#include <cstring>    //为了应用 strcmp()
#include <cstdlib>     //为了应用 EXIT 代码
#include <cassert>
using namespace std;
```

```
main(int argc, char *argv[])
{
    const int BUFSIZE = 128;
    char buf1[BUFSIZE], buf2[BUFSIZE];

    //打开文件:
    if (argc != 3)
        return EXIT_FAILURE;
    ifstream f1(argv[1]);
    ifstream f2(argv[2]);

    //做合并:
    f1.getline(buf1, BUFSIZE);
    f2.getline(buf2, BUFSIZE);
    while (f1 && f2)
    {
        // (不变式: 两个缓冲器都有新行)

        // 打印和刷新适当的行:
        if (strcmp(buf1, buf2) <= 0)
        {
            cout << buf1 << endl;
            f1.getline(buf1, BUFSIZE);
        }
        else
        {
            cout << buf2 << endl;
            f2.getline(buf2, BUFSIZE);
        }
    }
}
```

```

// (不变式: 至少有一个文件已经读完)

// 清空剩下的文件
while (f1)
{

    // (不变的: buf1 有一个新行)
    cout << buf1 << endl;
    f1.getline(buf1, BUFSIZE);
}
assert(f1.eof()); // (不变式: file1 已经读完)
while (f2)
{
    // (不变的: buf2 有一个新行)

    cout << buf2 << endl;
    f2.getline(buf2, BUFSIZE);
}
assert(f2.eof()); // (不变式: file2 已经读完)

return EXIT_SUCCESS;
}

// 示例执行结果:
c:>mergel file1.dat file2.dat
brown
dog
fox
jumped
lazy
over
quick
the
the

```

可以很容易地扩展合并算法以处理大量的文件。乍看之下可能会认为循环控制语句
while (两个文件其中一个没读完)

应该变为:

```
while (没一个文件被读完)
```

但如果有 n 个文件, 那么, 当一个循环结束时还有 $n-1$ 个现行的文件要继续合并。必须一直留在合并循环内直到只剩下一个文件可用:

```
while (可用文件数>1)
```

(等等)

这就要求有一份可用文件的列表, 当其中的文件用完输入行时可以在列表中删除这个条目。该设计如下:

```

while (现行文件数>1)
{
    决定接下来是那一行
    打印该行
    if (对应文件为空)
        从现行文件列表中删除
    else
        从这个文件 跳到新一行
}
清空剩余的文件

```

既然已经介绍了跟踪现行文件的机制，就只需呆在循环体中直到最后的文件为空，别的什么都不用做。程序清单 12.6 中的伪代码反映了这个变化，它和在程序清单 12.7 实现的不变条件一样。程序清单 12.7 中的程序用一系列指向 FileInfo 对象的指针来跟踪文件的状态，该对象包含了文本的当前行和指向其文件流的指针。构造函数 FileInfo 和函数 getNextLine 一起用来确保列表中的每个文件拥有一个新行的条件不变。

程序清单 12.6 合并任意数量的文件过程的伪代码

```

open and read first line from each file
while (a file with a fresh line remains)
{
    determine which lines comes next lexicographically
    print that line
    if (the corresponding file is now at end-of-file)
        delete it from the list of active files
    else
        get a fresh line from the file
}

```

程序清单 12.7 程序清单 12.6 的 C++ 实现

```

// merge2.cpp: 合并文件到标准输出
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
#include <list>
using namespace std;

const int BUFSIZE = 128;

struct FileInfo
{
    ifstream* f;
    char line[BUFSIZE];
    FileInfo(char* fname)

```



```

    {
        f = new ifstream(fname);
        f->getline(line, BUFSIZE);
    }
    ~FileInfo()
    {
        f->close();
        delete f;
    }
};

main(int argc, char *argv[])
{
    string getNextLine(list<FileInfo*>&);
    continued
    if (argc == 1)
        return EXIT_FAILURE;

    // 打开 files-read 首行:
    list<FileInfo*> flist;
    for (int i = 0; i < argc-1; ++i)
        flist.push_back(new FileInfo(argv[i+1]));

    // 做合并:
    while (flist.size() > 0)
        cout << getNextLine(flist) << endl;

    return EXIT_SUCCESS;
}

string getNextLine(list<FileInfo*>& flist)
{
    //按排序顺序查找下一行:
    char* next = "\xff";
    list<FileInfo*>::iterator small;
    list<FileInfo*>::iterator p = flist.begin();
    while (p != flist.end())
    {
        FileInfo* fp = *p;
        if (strcmp(next, fp->line) > 0)
        {
            small = p;
            next = fp->line;
        }
        ++p;
    }
}

```

```
    }

    //提升相应的文件流到新行:
    FileInfo* fp = *small;
    ifstream* f = fp->f;
    char* line = fp->line;
    string minLine(line);
    f->getline(line, BUFSIZE);
    if (f->eof())
    {

    // 关闭并删除入口:
        delete fp;
        flist.erase(small);
    }

    // (不变式: 每个 flist 入口仍有新行)
    return minLine;
}

// 示例执行结果:
c:>merge2 file1.dat file2.dat file3.dat
and
brown
cow
dog
fox
jumped
jumped
lazy
moon
over
over
quick
the
the
the
the
```

12.2 分支

即使可能不用 `goto` 进行编程, 在循环中间进行转移经常也是很方便的 (`goto` 语句的特殊情况)。应用 C++ 的 `break` 语句, 可以防止用外部的循环控制变量把代码弄乱。例如, 程序清单 12.2 中的变量 `done` 和 `found` 只是用来终止它们的循环。大多数 C++ 程序员不用它们 (见程序清单 12.8, 其中我用了 `break` 语句)。

程序清单 12.8 把外部的循环控制从程序清单 12.2 中删除

```

// hi-lo2.cpp: 移动外部布尔标志
#include <iostream>
#include <cctype>
using namespace std;

main()
{
    cout << "Think of a number between 1 and 100.\n"
        << "If you don't cheat, I'll figure it out\n"
        << "in seven guesses or less!"
        << endl;

    for (;;)          //无限循环的习惯用法
    {
        int lo = 1, hi = 100;
        char response;
        int guess;

        //开始游戏:
        while (lo <= hi)
        {

            //得到 guess 的值
            guess = (lo + hi) / 2;
            cout << "Is it " << guess << " (L/H/Y): " << endl;
            cin >> response;
            response = toupper(response); continued

            //缩小搜索范围:
            if (response == 'L')
                lo = guess + 1;
            else if (response == 'H')
                hi = guess - 1;
            else if (response != 'Y')
                cout << "What? Try again..." << endl;
            else
                break;
        }

        //打印结果:
        if (lo > hi)
            cout << "You cheated!" << endl;
        else
            cout << "Your number was " << guess << endl;
    }
}

```

```

        cout << "Wanna play again? " << endl;
        cin >> response;
        if (toupper(response) != 'Y')
            break;
    }
}

```

假设打算在程序清单 12.8 中把注释（//缩小搜索范围）下面的 if-else 语句用 switch 语句重写：

```

//缩小搜索范围
switch (response)
{
    case 'L':
        lo=guess+1;
        break;
    case 'H':
        hi=guess-1;
        break;
    case 'Y':
        puts("what? Try again !");
        break;
    default:
        break;    //这个不起作用!
}

```

不幸的是在默认状态下的 break 语句只退出 switch 语句，而不退出循环。这种情况下，可以像以前一样用 goto 语句或 if-else 结构。

用 goto 退出深层嵌套循环比用布尔值标志更清晰，更接近纯化论者所提倡的思想。考虑下面在一个三维数组中查找某个值的程序片断：

```

for(i=0;i<n;++i)
    for (j=0;j<m;++j)
        for(k=0;k<s;++k)
            if (a[i][j][k]==x)
                goto found;
found:

```

/*此处用 i、j 和 k*/

下面是一个天真的无分支方案的尝试：

```

for (done=0,i=0;!done&& i<n;++i)
    for(j=0;!done&& j<m;++j)
        for(k=0;!done&& k<s;++k)
            if(a[i][j][k]==x)
                done=1;
/*固定索引(讨厌!)*/
--i,--j,--k;

```

就算不介意这种方法，它自始至终把控制传递给封闭循环的顶层而不是马上退出，这在当前的值以外增加了循环变量。

将 `goto` 用作多级循环的出口能在出现严重错误之后帮助程序恢复到稳定状态，如：

```
main()
{
    //这里是以前的初始化
    for(;;)
    {
        //这里是更多的初始化
        //深层嵌套
        if (<无法控制>)
            goto recover;
    }
recover:
}
```

注意标记后跟有分号。标记不能独自存在，通常必须作为标记语句的一部分存在。

12.3 非局部分支

如果遇到一个异常的条件，它处于一系列嵌套函数调用的深处，并且还原指针处于调用链中一个函数的高层（如 `main`），这时该怎么办呢？所需的就是一个转移跨过函数的高级 `goto`。就是 `setjmp/longjmp` 结构所要做的。用 `setjmp` 来记录返回点，并用 `longjmp` 转移到这点。其句法是：

```
# include <setjmp.h>
jmp_buf recover;
main()
{
    volatile int i=0;
    for(;;)
    {
        if (setjmp(recover)!=0)
        {
            /*从 f() 中的错误恢复*/
        }
        /*深层嵌套*/
    }
    return 0;
}
...
void f()
{
    /* 做一些危险的事情*/
    if (<无法控制>)
```

```

longjmp(recover, 1);
/* 否则继续进行 */
}

```

`jmp_buf` 是一个数组，它保存把执行恢复到 `setjmp` 点所必需的系统信息。（显然，跳转缓冲区必须是全局的）。当调用 `setjmp` 时，系统在 `recover` 中存储调用环境（例如栈和指令指针寄存器的内容等等）。当直接调用时，`setjmp` 通常的返回值是零。`Longjmp` 的调用则恢复调用环境，所以执行继续回到 `setjmp` 调用，但有一点不同：看起来似乎 `setjmp` 回到 `longjmp` 调用中的第二个参数（在这种情况下是 1）。如果把 `longjmp` 的第二个参数赋值为零，则它不管在什么情况下都返回 1。由于 `longjmp` 完成了从一个函数的交替返回，因此它打断了一个程序的正常运行，因此应该只在处理特殊情况时使用 `longjmp`。

含有 `setjmp` 的目标函数必须还要是可用的（即它必须还没有返回到它的调用点），否则该调用环境将不再是有用的。当然，同一个 `jmp_buf` 变量有多个 `setjmp` 终点可不是一个好的想法。（因为那会使人感到困惑！）由于典型的调用环境包含了寄存器，并且一个编译器能自由地在寄存器中存储自动变量，因此从 `longjmp` 返回时，没有办法知道一个自动的对象是否有正确的值。应该在任何包含 `setjmp` 的函数中用 `volatile` 限定词来声明自动变量，这样可以确保 `longjmp` 的内部工作不被打扰。

在一个 `longjmp` 中不可以相信自动存储的另外一个结论是：调用自身的 `setjmp` 决不能在可能产生一个临时对象的表达式中出现，例如一个中间的计算。出于这个原因，`setjmp` 的调用作为一个单一控制表达式只能在 `if` 或 `switch` 语句中出现，或作为一个操作数出现在相应的表式中。不能把 `setjmp` 的结果可靠地赋值给其他变量。程序清单 12.9 中的程序删除了子程序清单树，在 `switch` 语句中调用 `setjmp`（关于更多细节参见第 19 章目录 I/O 函数）。命令行变量 `argc` 和 `argv` 被声明为 `volatile`，因为它们存储在一个包含 `setjmp` 的函数的堆栈中。在 `main` 后半部分的 `char*` 的两个强制类型转换在调用它们各自的函数之前去掉了 `volatile`。函数 `rd` 中的循环说明了 `continue` 语句，该语句跳到下一个循环迭代。程序清单 12.7 的响应文件需要为 MS-DOS 的 `del` 命令的特殊行为做补充，这个命令需要一个关键字响应来删除目录中所有的文件。

程序清单 12.9 说明非局部分支的递归目录删除程序

```

/* ddir.c: 移动子目录树 */
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <string.h>
#include <sys/stat.h>
#include <setjmp.h>
#include <dirent.h>
#include <dir.h>

/* longjmp 返回的代码 */
#define BAD_DIR 1

```

```
#define DIR_OPEN_ERR 2
#define FILE_DEL_ERR 3
#define DIR_DEL_ERR 4
    continued
/* DOS 操作系统与其他的操作系统具体的宏变化 */
#define CMD_FORMAT "del *.* <%s > nul"
#define CMD_LEN 17

char Response_file[L_tmpnam+1] = "/";

/* 调用环境缓冲器 */
jmp_buf env;

main(volatile int argc, volatile char **argv)
{
    FILE *f;
    volatile char *old_path = getcwd(NULL, 64);
    void rd(char *);

    /* 为 DOS 系统 del 命令创建响应文件 */
    tmpnam(Response_file+1);
    if ((f = fopen(Response_file, "w")) == NULL)
        abort();
    fputs("Y\n", f);
    fclose(f);

    switch(setjmp(env))
    {
        case BAD_DIR:
            fputs("Invalid directory\n", stderr);
            break;
        case DIR_OPEN_ERR:
            fputs("Error opening directory\n", stderr);
            break;
        case FILE_DEL_ERR:
            fputs("Error deleting file\n", stderr);
            break;
        case DIR_DEL_ERR:
            fputs("Error deleting directory\n", stderr);
            break;
    }

    /* 删除目录 */
    while (--argc)
        rd((char *) *++argv);
}
```

```
/* 清除 */
remove(Response_file);
chdir((char *) old_path);
return 0;
}

void rd(char* dir)
{
    char sh_cmd[L_tmpnam+CMD_LEN];
    DIR *dirp;
    struct dirent *entry;
    struct stat finfo;

    /* 登录将要被删除的目录 */
    if (chdir(dir) != 0)
        longjmp(env, BAD_DIR);
    printf("%s:\n", strlwr(dir));

    /* 通过 OS 外壳删除所有的一般文件 */
    sprintf(sh_cmd, CMD_FORMAT, Response_file);
    system(sh_cmd);

    /* 删除任何保留的目录入口 */
    if ((dirp = opendir(".")) == NULL)
        longjmp(env, DIR_OPEN_ERR);
    while ((entry = readdir(dirp)) != NULL)
    {
        if (entry->d_name[0] == '.')
            continue;
        stat(entry->d_name, &finfo);
        if (finfo.st_mode & S_IFDIR)
            rd(entry->d_name); /* Subdirectory */
        else
        {
            /* 允许删除文件，然后完成删除 */
            chmod(entry->d_name, S_IWRITE);
            if (unlink(entry->d_name) != 0)
                longjmp(env, FILE_DEL_ERR);
        }
    }
    closedir(dirp);

    /* 从父目录中删除目录 */
    chdir("..");
    if (rmdir(dir) != 0)
        longjmp(env, DIR_DEL_ERR);
}
```


12.4 信号

当一个不寻常的事件打断程序的正常运行时，例如，把零当作除数或用户按下了注意信号键（例如，Control-C 或 Del），会发出一个信号。头文件<signal.h>定义了下列“标准信号”：

SIGABRT	非正常终止（由 abort 发出）
SIGFPE	计算的异常（例如，溢出）
SIGILL	无效的函数映射（例如，非法指令）
SIGINT	交互式的注意信号（例如，Control-C）
SIGSEGV	试图访问受保护的内存
SIGTERM	终止请求

这些信号起源于 UNIX 下的 PDP 结构，不一定都适用于你的环境。实现也可以定义其他的信号或完全忽略信号，因此信号处理自然是不可移植的。

信号分成两种：同步和异步。同步信号是由程序发出的，例如以零为除数、一个浮点溢出操作或是发出对 abort 函数调用。可以通过调用 raise 函数来显式地发出一个信号，例如：

```
raise(SIGABRT)
```

异步信号是由程序不能预见的外部力量所产生的，例如用户按下注意信号键。

大多数信号的默认响应通常是终止程序，但是既可以忽略此信号也可以提供自己惯用的信号处理程序。下列语句出自程序清单 12.10。

```
signal(SIGINT, SIG_DFL)
```

它告诉环境忽略任何键盘的中断请求（在我的机器上为 Control-C）。键盘输入过程仍然在显示器上回显^C 标志，但是它不把控制传递给终止程序的默认处理信号逻辑。语句：

```
signal(SIGINT, SIG_DFL)
```

恢复了默认行为，这样就能够用键盘使程序终止。

程序清单 12.11 的程序调用库函数 signal 来安装函数 abort_handler 以响应 SIGABRT。一个信号处理程序必须带有一个整型参数并返回 void。abort 的第一次调用传递控制权给 abort_handler，把 SIGABRT 作为整型实参传递（这里我不是碰巧使用），然后打印消息。

```
abort signal intercepted
```

程序清单 12.10 关闭键盘中断请求

```
/* ignore.c: 忽略交互式的注意信号键 */
#include <stdio.h>
#include <signal.h>

main()
{
    char buf[BUFSIZE];
    int i;
```

```

/* 忽略键盘中断 */
signal(SIGINT, SIG_IGN);

while (gets(buf))
    puts(buf);

/* 恢复默认注意键处理
   因而使用者能由键盘进行异常中断*/
signal(SIGINT, SIG_DFL);
for (i = 1; ; ++i)
    printf("%d%c", i, (i%15 ? ' ' : '\n'));
}

```

/* 示例执行结果

```

c:>ignore
hello
hello
^C
there
there
^C
^Z
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55 56 57 58 59 60
61 62 63 64 65 66 67 ^C
*/

```

当操作进入一个信号处理程序时，环境就认为在第一个语句执行之前关联的信号已经被“处理过了”。这样就打破了处理程序和信号之间随后的任何联系，然后恢复默认处理。这意味着如果处理程序返回了，那么在下一信号出现时它就不会取回控制权。这就是为什么当控制回到它在 `main` 中被中断的地方并第二次调用 `abort` 时，程序实际上被终止了。实际上，一个中断的处理程序不应返回，而应该调用 `exit` 来终止程序。同样的，如果从一个 `SIGFPE` 处理程序中返回值，结果是无法确定的。

程序清单 12.11 截取 `SIGABRT(abort)` 信号

```

/* abort.c: 句柄 SIGABRT */
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void abort_handler(int sig);

```

```

main()
{

    /* 设置信号处理函数 */
    if (signal(SIGABRT, abort_handler) == SIG_ERR)
        fputs("Error installing abort handler\n", stderr);

    abort();
    abort();
    return 0;
}

void abort_handler(int sig)
{
    fputs("Abort signal intercepted\n", stderr);
}

/* 输出:
Abort signal intercepted
Abnormal program termination
*/

```

如果想在—个信号每次发生时都处理它，处理程序应该在每次执后重新设置自己的信号。如在程序清单 12.12 中的 SIGINT 处理程序 (ctrlc) 所示，应该在异步信号处理程序的最开始时就这么做。如果在函数中推迟稍许，同样的信号在重新设置它之前就出现了，这会导致丢失系统默认处理程序的信号。实际上，异步信号的处理很少会是安全的。正在处理的信号可能已在库函数的执行中发生了。既然标准 C 库没能保证在所有的平台上都可重入，那么在处理程序中调用该库函数就不是一个好主意。规则是：不要在一个处理异步信号的函数中调用库函数。

程序清单 12.12 一个安全的用来计数键盘中断的 SIGINT 处理程序

```

/* ctrlc.c: 一个安全的 SIGINT 处理函数 */
#include <stdio.h>
#include <signal.h>

void ctrlc_handler(int sig);

volatile sig_atomic_t ccount = 0;

main()
{
    char buf[BUFSIZE];

    /* 设置 SIGINT 处理函数 */

```

```
    if (signal(SIGINT,ctrlc_handler) == SIG_ERR)
        fputs("Error installing ctrlc handler\n",stderr);

    /* 做一些输入 / 输出处理 */
    while (gets(buf))
        puts(buf);

    /* 恢复默认处理函数 */
    signal(SIGINT,SIG_DFL);
    printf("You pressed ctrlc %d times\n",ccount);
    return 0;
}

void ctrlc_handler(int sig)
{
    /* 立即重新设置处理函数 */
    signal(sig,ctrlc_handler);

    ++ccount;
    return;
}

/* 示例执行结果
c:>ctrlc
hello
hello
^C
there
there
^C
^Z
You pressed ctrlc 2 times
*/
```

程序清单 12.13 一个命令解释程序的框架——说明转移到信号处理程序之外的分支

```
/* shell.c: 命令解释器的框架 */
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
/* 等等 */

jmp_buf restart;

void ctrlc(int sig);
```

```

{

/* 设置信号处理函数 */
    signal(SIGINT,ctrlc);

/* 键盘中断返回到这里 */
    setjmp(restart);

    /* 在这里做任何其他的初值设定... */
    for (;;)
    {
        int command_code;
        /* 等等 */

        /* 在此读和解析命令... */
        /* 在此执行内部命令 */
        switch(command_code)
        {
            case 'Q':
                return 0; /* 退出 */

            /* 接下来可以发生许多情况 */
        }
    }
}

void ctrlc(int sig)
{

    /* 重新设置处理函数 */
    signal(sig,ctrlc);

    /* 跳回到命令行 */
    longjmp(restart,1);
    return;
}

```

不应该调用任何发出另一个信号的函数。如 `ctrlc` 函数所示，在从异步的处理程序返回之前你只能做两件安全的事：

1. 调用信号重新设置处理程序。
2. 为 `volatile sig_atomic_t` 类型的静态对象赋值。

`sig_atomic_t` 类型是一个完整的类型，它能保证当信号被访问时不被另一个信号所干扰。在正常运行重新开始时可以把这个值当作一个标志或者是一个计数器。尽管是多余的，但在处理程序尾部添加一个明确的 `return` 语句也是好的，这样可以区别那些使程序停止的处理程序。

不管前面一段怎么说,在许多环境里用 `longjmp` 跳转出异步信号处理程序仍然是常规方法。(在 DOS 系统和 UNIX 的多数时间里 `longjmp` 都能正常工作。)举一个普通的例子,程序清单 12.13 是命令行注释程序的框架,该程序读取一文本序列指令并且完成显示功能(与 UNIX 中的 `sh` 或者 DOS 的 `COMMAND.COM` 类似)。如果在深层的内部逻辑中按下注意信号键,是想返回到命令循环,而不是想终止程序。

12.5 小结

- 任何算法都可以用顺序语句、选择语句、循环语句和一个任意数量的布尔型变量来表示。
- 用 `if-else` 或 `switch-case` 来实现选择。
- 用 `while`、`for`、或 `do-while` 实现循环。
- 用断言或注释显示不变的条件。
- `break` 退出密封关闭的循环或分支。
- `continue` 跳到最近关闭的循环的下一个迭代。
- 只能用 `goto` 来跳出嵌套循环。
- 用 `setjmp/longjmp` 机制在异常条件下交替地返回。
- 用信号处理程序捕获同步信号和异步信号。

12.6 参考文献

1. Edsger Dijkstra, *GOTO Statement Considered Harmful*, CACM 11:3 (1968 年 3 月), 147。
2. Edsger Dijkstra, *Stepwise Program Construction*, 发表于 *Selected Writings on Computing: A Personal perspective* (Springer-Nerlag, 1982), 2 页。
3. C.Bohm 和 G.Jacopini, *Flow Diagrams, turing Machines, and languages with Only Two Formation Rules*, CACM 9:5 (1966 年 5 月), 226 页。
4. P.J.Plauger, *Programming on Purpose, Essays on Software Design* (Englewood Cliffs, NJ:PrenticeHall, 1993), 25 页。

异 常

13.1 可选择的错误处理方法

在以往传统的程序设计语言中，开发者可选择的错误处理方法主要由下面几种组成：

1. 忽略它们；
2. 检查返回代码（一丝不苟地）。

第一种方法只是在无关紧要的不讨论异常的程序中使用，如学生作业或是杂志文章。这种无策略可言的方法在要真正执行时起不了任何作用。例如，考虑程序清单 13.1 中的程序，它用标准 POSIX 目录处理程序来删除整个目录。（关于这些函数的解释见第 18 章）。什么可能会出错？喔，我看到有下边几种可能：

- a. 被选择的目录并不存在。
- B. 目录是读保护的。
- C. 目录中的文件是删除保护的。
- D. 目录本身是删除保护的。

程序清单 13.1 一个删除整个目录子树的程序

```
/* ddir.c: 删除子目录树 */  
#include <stdio.h>  
#include <stdlib.h>  
#include <io.h>  
#include <string.h>  
#include <sys/stat.h>  
#include <dirent.h>  
#include <dir.h>
```

```
main(int argc, char **argv)
{
    char *old_path = getcwd(NULL, 64);
    void rd(char *);

    /* 删除目录 */
    while (--argc)
        rd(++argv);

    /* 恢复目录 */
    chdir(old_path);
    return 0;
}

void rd(char* dir)
{
    void erase_dir(void);

    /* 记录将删除的目录 */
    chdir(dir);

    /* 经 OS 外壳 (DOS 系统版本) 删除全部的正规文件 */
    system("del /q *.* > nul");

    /* 删除所有保留的目录入口 */
    erase_dir();

    /* 从父目录中去掉目录 */
    chdir("..");
    rmdir(dir);
}

void erase_dir()
{
    DIR *dirp;
    struct dirent *entry;
    struct stat finfo;    continued
    /* 删除当前目录 */
    dirp = opendir(".");
    while ((entry = readdir(dirp)) != NULL)
    {
        if (entry->d_name[0] == '.')
            continue;
        stat(entry->d_name, &finfo);
        if (finfo.st_mode & S_IFDIR)
```



```

        rd(entry->d_name);      /*子目录*/
    else
    {

        /* 允许删除文件，然后实现 */
        chmod(entry->d_name, S_IWRITE);
        unlink(entry->d_name);
    }
}
closedir(dirp);
}

```

出于这个原因，大多数 C 库函数返回一个状态码，如果有错误发生则可以通过测试状态码来发现错误。如程序清单 13.2 所示，这将需要在一个深层的嵌套调用链中做许多检查——嵌套越深，你会越累。由于在这种情况下的目标是使程序返回到安全状态，如果有种办法能把执行过程从深层嵌套调用链中拉出来，并把它放在调用链的上层就好了。这种机制是存在的，即为第三种错误处理方法：

3. 用非局部跳转来使执行过程改变方向。

这就是 C 中的 `setjmp/longjmp` 机制，如第 12 章所示（见程序清单 12.9）。`setjmp` 函数使用了一个跳转缓冲区（`jmp_buf`），它是一个编译器定义的结构，这个结构记录了足够的信息来恢复对程序中前一点的控制。当 `setjmp` 第一次执行时，它初始化缓冲区并返回 0。如果一个 `longjmp` 在涉及相同 `jmp_buf` 的调用链中调用得很深，那么控制转移将立即回到 `setjmp` 调用，这好像从 `longjmp` 调用中返回了第二个参数。如果经 `longjmp` 调用返回 0 那将会是真正的麻烦事（我认为你是永远不会这样做的）。所以如果其他一些愚蠢的程序员这样做了，`setjmp` 无论如何也将返回一个 1。

看起来所有的问题都已经被解决了，我现在可以和你再说再见了。然而，事实证明在 C++ 中由一个函数跳转到调用链中另外的上层可能是危险的。如程序清单 13.3 所示，主要的问题是在 `setjmp` 和 `setjmp` 之间的执行过程中创建的自动变量将无法合理地撤消。幸运的是，C++ 通过异常处理机制解析构造函数从而提供这样的交替返回。所以，如果在 C++ 环境下编程，你可以考虑这种最后的错误处理方法：

4. 使用异常

程序清单 13.2 与程序清单 13.1 相似但有返回代码

```

/* ddir2.c: 删除子目录树 */
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <string.h>
#include <sys/stat.h>
#include <dirent.h>
#include <dir.h>

```

```
/* 返回代码 */
#define BAD_DIR 1
#define DIR_OPEN_ERR 2
#define FILE_DEL_ERR 3
#define DIR_DEL_ERR 4

main(int argc, char **argv)
{
    char *old_path = getcwd(NULL, 64);
    int rd(char *);

    while (--argc)
    {
        int code = rd(++argv);
        switch(code)
        {
            case BAD_DIR:
                puts("Invalid directory");
                break;
            case DIR_OPEN_ERR:
                puts("Error opening directory");
                break;
            case FILE_DEL_ERR:
                puts("Error deleting file");
                break;
            case DIR_DEL_ERR:
                puts("Error deleting directory");
                break;
        }
    }

    chdir(old_path);
    return 0;
}

continued
int rd(char* dir)
{
    int erase_dir(void);
    int code;

    if (chdir(dir))
        return BAD_DIR;
    system("del /q *.* > nul");

    code = erase_dir();
    if (code)
```

```

        return code;

        chdir("..");
        if (rmdir(dir))
            return DIR_DEL_ERR;
        return 0;
    }

int erase_dir()
{
    DIR *dirp;
    struct dirent *entry;
    struct stat finfo;

    if ((dirp = opendir(".")) == NULL)
        return DIR_OPEN_ERR;

    while ((entry = readdir(dirp)) != NULL)
    {
        if (entry->d_name[0] == '.')
            continue;
        stat(entry->d_name, &finfo);

        if (finfo.st_mode & S_IFDIR)
            rd(entry->d_name);
        else
        {
            chmod(entry->d_name, S_IWRITE);
            if (unlink(entry->d_name))
            {
                closedir(dirp);
                return FILE_DEL_ERR;
            }
        }
    }
    closedir(dirp);
    return 0;
}

```

程序清单 13.3 展示 longjmp 留下的未被撤销的对象

```

// destroy1.cpp
#include <iostream>
#include <csetjmp>
using namespace std;

class Foo

```

```
{
public:
    Foo() {cout << "Foo constructor\n";}
    ~Foo() {cout << "Foo destructor\n";}
};

jmp_buf env;

main()
{
    void f();
    if (setjmp(env) == 0)
        f();
    else
        cout << "Returned via longjmp" << endl;
    return 0;
}

void f()
{
    void g();
    Foo x;
    g();
}

void g()
{
    Foo x;
    longjmp(env, 1);
}

// 输出:
Foo constructor
Foo constructor
Returned via longjmp
```

13.2 堆栈展开

程序清单 13.4 中的程序使用异常对程序清单 13.3 中的程序进行重写。可以在代码段中通过用 `try` 块包围异常来打开异常捕捉。在 `try` 块后立即加入异常处理程序。异常处理程序就像用关键字 `catch` 命名的函数定义。可以用 `throw` 表达式抛出异常。语句:

```
throw 1;
```

使编译器在嵌套函数调用链中倒退查找一个可以捕捉整数的处理程序, 这样在 `main` 中控制就

转到该处理程序中。运行期环境在执行了进入 `try` 块后为所有构造的自动对象调用析构函数。这个在异常处理程序途中撤销自变量的过程叫做堆栈展开。

正如所看到的，`throw` 表达式的执行在语义上与调用 `longjmp` 类似，处理程序也很像 `setjmp`。只是不可以长跳转（`longjmp`）到已经返回了的函数，而只能跳转到与“现行的”`try` 块相关的处理程序，即尚未执行的程序（这种情况将返回 0）。

异常与 `setjmp / longjmp` 功能的一些显著的区别是：

1. 异常处理是一种语言机制而不是函数库特性。转换控制到处理程序的额外开销对于程序员是不可见的。
2. 编译器跟踪所有具有析构函数的自动变量并在必要时执行这些析构函数（即展开堆栈）。
3. 函数中的那些包含 `try` 块的局部变量是“安全”的——没有必要像在 `setjmp` 那样把它们声明为 `volatile` 以保证它们不被讹误。
4. 通过匹配所抛出的异常对象的类型来找到处理程序。这允许用单独的处理程序来处理各种异常并通过继承来给它们分类。
5. 异常处理是一个运行期机制。通过检测源代码无法总是断定哪一个处理程序可以捕捉一个指定的异常。

程序清单 13.4 说明堆栈展开

```
// destroy2.cpp
#include <iostream>
using namespace std;

class Foo
{
public:
    Foo() {cout << "Foo constructor" << endl;}
    ~Foo() {cout << "Foo destructor" << endl;}
};

main()
{
    void f();
    try          //开启异常处理
    {
        f();
    }
    catch(int)
    {
        cout << "Caught exception" << endl;
    }
    return 0;
}
```

```
void f()
{
    void g();
    Foo x;
    g();
}

void g()
{
    Foo x;
    throw 1;
}

// 输出:
Foo constructor
Foo constructor
Foo destructor
Foo destructor
Caught exception
```

最后一点有重要意义。C++异常处理机制允许在错误检测和错误处理之间有清楚的分离。一个函数库开发者能检测到什么时候会出现错误，如参数不在范围之内，但他也许不知道该怎么处理它。你作为一个用户虽然不可能总是检测异常情况，但你知道你的应用程序需要怎样去处理它。因此，异常为应用程序组件间的运行期错误通信制定了协议。

认识到 C++异常处理是围绕中断模式设计的也同样重要，就是说当异常被抛出时没有直接的办法像在遵循继续执行模式的 Ada 语言中所做的那样——回到抛出点并在被中断的地方继续执行程序。C++异常是为不常发生的同步事件服务的。

13.3 异常捕捉

由于异常是一个运行期而不是编译期特性，标准 C++为异常与捕捉参数相匹配制定了一些规则，它与为找到重载函数以匹配函数调用而制定的规则有点不同。可以用下面的方法之一为 T 类型对象定义一个处理程序（变量 t 是任意的，就像在 C++中不同的函数参数一样）：

```
catch(T t)
catch(const T t)
catch(T& t)
catch(const T& t)
```

这种处理程序可以捕捉到 E 类型的异常对象，如果：

1. T 和 E 是同一类型，或
2. 在抛出点上 T 是 E 的可访问基类，或

3. T 和 E 是指针类型并且在抛出点上 E 与 T 间存在一个标准指针类型转换。

如果从 E 到 T 的所有公有派生类间存在一条从 E 到 T 的继承路径, 则 T 是 E 的可访问基类。为了理解第 3 条, 假设类型 E 指向类型 F, 类型 T 指向类型 U。那么在 E 到 T 之间将存在一个标准指针转换, 如果:

1. T 与 E 是同种类型, 只是它可能已添加了 `const` 和 `volatile` 限定词, 或
2. T 是一个 `void*` 类型, 或
3. U 是明确的 F 的可访问基类, 这意味着 F 的成员可以明确地引用 U 的成员 (通常只关心多重继承)。

这些规则的底线是异常和捕捉参数必须相互准确地匹配, 或者由指针或引用捕捉的异常必须是从捕捉参数的类型派生来的。例如下面的异常就没被捕捉到:

```
#include <iostream>
using namespace std;

void f ( );
main( )
{
    try
    {
        f ( );
    }
    catch (long)
    {
        cerr <<"caught a long"<<endl;
    }
}

void f ( )
{
    throw 1; //不是一个长整型
}
```

当系统找不到异常的处理程序时将调用标准库函数 `terminate`, 这个函数在默认情况下将终止程序。可以通过把它作为 `set_terminate` 的参数来替代自己的终止函数, 如下面的程序所示:

```
//terminat.cpp
#include <iostream>
#include<exception>
#include<cstdlib>
using namespace std;

void handler( )
{
    cout<<"Renegade exception!\n";
    exit(EXIT_FAILURE);
}
```

```
void f ( );
main( )
{
    set_terminate(handler);
    try
    {
        f ( );
    }
    catch (long)
    {
        cerr <<"caught a long"<<endl;
    }
}
void f ( )
{
    throw 1;
}
```

//输出:

Renegade exception!

下面的异常能被捕捉到, 因为存在一个可访问基类的处理程序:

```
#include <iostream>
class B {};
class D : public B {};
void f ( );
main( )
{
    try
    {
        f ( );
    }
    catch (const B&)
    {
        cerr <<"caught a B"<<endl;
    }
}
void f ( )
{
    throw D{};
}
```

被指针捕捉的异常也能被 void* 型处理程序所捕捉。

问题: 当控制转到处理程序时 throw 语句的内容将丢失, 怎样才能在处理程序中得到异常对象呢?

答案: 这是一个很好的问题! 这是异常处理与函数调用的又一不同之处。运行期机制产生一个抛出对象的临时拷贝为处理程序所用。这意味着用值语义定义捕捉参数永远不会是一

个好的办法，因为它又将生成第二个拷贝。同时，把一个捕捉参数定义为指针也是危险的，因为无法知道它是否来自堆栈中（这种情况必须删除它）。最好的策略是一直采用引用捕捉。即使异常是临时的，如果需要的话也可以把它当成非常数引用来捕捉（然而从一个正常函数处理规则的另一个出发点看，推荐使用异常处理）。

13.4 标准异常

标准 C++ 库抛出的不同类型的异常对象可以从下面的类继承找到：

```
exception
    logic_error
        domain_error
        invalid_argument
        length_error
        out_of_range
    runtime_error
        range_error
        overflow_error
        underflow_error
    bad_alloc
    bad_cast
    bad_exception
    bad_typeid
```

逻辑错误表示程序内部逻辑有矛盾，或是违背了客户端软件的前提条件。例如，若要求子字符串开始处超过字符串的结尾时，标准字符串类中的 `substr` 成员函数便抛出一个 `out_of_range` 异常。运行时错误是很难被预知的，并且经常是由于程序外部引起的。例如，范围错误，即违反了函数的后置条件，如处理合法参数时的算术溢出。

在堆栈资源用尽时发生 `bad_alloc` 异常（见下面的“内存管理”）。当 `dynamic_cast` 引用类型失败时 C++ 将生成一个 `bad_cast` 异常。如果从一个 `unexpected` 处理程序内重新抛出异常，它将隐藏于一个 `bad_exception` 异常中（见下面的“异常规范”）。如果试图将 `typeid` 操作符应用于空表达式中，则会得到一个 `bad_typeid` 异常。

程序清单 13.5 中的程序从 `runtime_error` 中派生得到 `dir_error` 异常类，因为关联的错误被系统提供的返回状况所检测。`exception` 基类有一个名为 `what` 的成员函数，它返回创建异常的字符串参数。我用这个函数把关于错误的信息传送到异常处理程序中。带省略号的处理程序（`catch (...)`）可以捕捉任何异常，所以程序文本中处理程序的次序就很重要：应该总是按照各自的类型从特殊到一般将它们排序。

13.5 资源管理

尽管堆栈展开负责删除自动变量，但还是会有一些需要单独删除。例如，考虑下面处理

文件的函数：

```
void f (const char* fname)
{
    FILE* fp=fopen(fname, "r");
    if (fp)
    {
        //处理文件；这里抛出异常只是为了说明；
        throw 1;    //强制抛出异常
        fclose(fp);    //这不会发生
    }
}
```

如果异常发生在 `f` 的中间时，文件将无法被关闭。能保证资源分配的方法之一是捕捉分配发生点所在函数中的所有异常。程序清单 13.6 中 `f` 里的处理程序将文件关闭然后通过重新抛出异常来传递在线上发生的任何异常（这是无参数的 `throw` 所做的事情）。然而，当在许多地方都用到相同的资源时，这种技术将是冗长乏味的。

程序清单 13.5 说明异常（exception）类

```
// ddir3.cpp: 删除子目录树
#include <cstdio>
#include <cstdlib>
#include <io.h>
#include <cstring>
#include <sys/stat.h>
#include <csetjmp>
#include <dirent.h>
#include <dir.h>

#include <stdexcept>
#include <string>
using namespace std;

//异常类
struct dir_error : public runtime_error
{
    dir_error(const string& msg) : runtime_error(msg){}
};

main(int argc, char** argv)
{
    char* old_path = getcwd(NULL, 64);
    void rd(const char*);

    while (--argc)
    {
        try
```

```

        {
            rd((const char*) *++argv);
        }
        catch (const dir_error& x)
        {
            printf("Directory error: %s\n",x.what());
        }
        catch (const exception& x)
        {
            printf("Error: %s\n",x.what());
        }
        catch (...)
        {
            puts("Unknown error");
        }
    }
    continued

// 清除 :
    chdir(old_path);
}

void rd(const char* dir)
{
    if (chdir(dir) != 0)
        throw dir_error("Invalid directory");
    system("del /q *.* > nul");

    erase_dir();

    chdir("..");
    if (rmdir(dir) != 0)
        throw dir_error("Error deleting directory");
}

void erase_dir()
{
    if ((dirp = opendir(".")) == NULL)
        throw dir_error("Error opening directory");
    struct dirent* entry;

    while ((entry = readdir(dirp)) != NULL)
    {
        if (entry->d_name[0] == '.')
            continue;
        struct stat finfo;

```

```
    stat(entry->d_name, &finfo);
    if (finfo.st_mode & S_IFDIR)
        rd(entry->d_name);
    else
    {
        chmod(entry->d_name, S_IWRITE);
        if (unlink(entry->d_name))
        {
            closedir(dirp);
            throw dir_error("Error deleting file");
        }
    }
}
closedir(dirp);
}
```

程序清单 13.6 在处理异常中分配资源

```
// destroy4.cpp
#include <cstdio>
using namespace std;

main()
{
    void f(const char*);
    try
    {
        f("file1.dat");
    }
    catch(int)
    {
        puts("Caught exception");
    }
}

void f(const char* fname)
{
    FILE* fp = fopen(fname, "r");
    if (fp)
    {
        try
        {
            throw 1;
        }
        catch(int)
        {
        }
    }
}
```

```

        fclose(fp);
        puts("File closed");
        throw; //重新抛出
    }

    fclose(fp); //正规的关闭
}

// 输出:
File closed
Caught exception

```

一个管理资源更好的方法可以让堆栈展开自动地分配资源。在程序清单 13.7 中我创建了一个局部类 `File`，它的构造函数打开文件而析构函数关闭文件。由于 `File` 对象 `x` 是自动的，因此它的析构函数在堆栈展开时可以保证自动被执行。可以用这个技术安全地处理任何数量的资源。

程序清单 13.7 说明“资源分配是初始化”的原则

```

// destroy5.cpp
#include <cstdio>
using namespace std;

main()
{
    void f(const char*);
    try
    {
        f("file1.dat");
    }
    catch(int)
    {
        puts("Caught exception");
    }
}

void f(const char* fname)
{
    class File
    {
        FILE* f;

    public:
        File(const char* fname, const char* mode)

```

```

        {
            f = fopen(fname, mode);
        }
        ~File()
        {
            fclose(f);
            puts("File closed");
        }
    };
    File x(fname, "r");
    throw 1;
}

// 输出:
File closed
Caught exception

```

13.6 构造函数和异常

向语言中加入异常的目的之一是为了弥补构造函数无返回值这一不足之处。在异常这个概念出现之前，在构造函数中处理资源错误是很棘手的。例如，在程序清单 13.8 中当对 `fopen` 的调用失败时会发生什么？一种解决方法是在使用资源前用内部状态变量进行检测。在程序清单 13.9 中的程序用从 `fopen` 返回的文件指针来决定资源是否可用。如果一切正常的话 `void*` 类型转换操作符返回一个非 0 的指针值，因此可以像下面这样来检测状态：

```

if (x)
    //一切正常
else
    //资源不可靠

```

程序清单 13.8 使用内部状态变量跟踪资源

```

// destroy6.cpp
#include <cstdio>
using namespace std;

main()
{
    void f(const char*);
    try
    {
        f("file1.dat");
    }
    catch(int)
    {

```

```

        puts("Caught exception");
    }
}

void f(const char *fname)
{
    class File
    {
        FILE* f;
    public:
        File(const char* fname, const char* mode)
        {
            f = fopen(fname, mode);
        }
        ~File()
        {
            if (f)
            {
                fclose(f);
                puts("File closed");
            }
        }
        operator void*() const
        {
            return f ? (void *) this : 0;
        }
    };

    File x(fname, "r");
    if (x)
    {
        //在此使用文件
        puts("Processing file...");
    }
    throw 1;
}

// 输出:
Processing file...
File closed
Caught exception

```

程序清单 13.9 在构造函数中抛出异常

```

// destroy7.cpp
#include <cstdio>

```

```
using namespace std;

class File
{
    FILE* f;

public:
    File(const char* fname, const char* mode)
    {
        f = fopen(fname, mode);
        if (!f)
            throw 1;
    }
    ~File()
    {
        if (f)
        {
            fclose(f);
            puts("File closed");
        }
    }
};

main()
{
    void f(const char*);
    try
    {
        f("file1.dat");
    }
    catch(int x)
    {
        printf("Caught exception: %d\n", x);
    }
} continued

void f(const char* fname)
{
    File x(fname, "r");
    puts("Processing file...");
    throw 2;
}

// 输出:
Processing file...
File closed
```


Caught exception: 2

然而，在使用每一个对象之前都对它进行检测是枯燥的。如程序清单 13.9 所示，由构造函数直接抛出异常可能会更方便。

只要你的 File 对象是自动的所有的事都会变得简单，但当从自由存储单元中分配 File 对象时会怎么样呢？如程序清单 13.10 所示，由于 File* 只是一个指针，所以没有析构函数被调用。由于在 C++ 中分配动态对象是很普遍的，因为标准库提供了 auto_ptr——一个有下面接口的模板类：

```
template<class T>class auto_ptr
{
public:
    explicit auto_ptr (T*=0);
    auto_ptr(auto_ptr<T>&);
    void operator=( auto_ptr<T>& r);
    ~ auto_ptr( );           //对基本指针调用删除
    T& operator*( );         //调用 T:: operator*
    T* operator->( );         //调用 T:: operator->
    T* get( ) const;         //返回基本的指针
    T* release( );           //失去指针
    T* reset(T*=0);          //首先释放原来的所有者
```

程序清单 13.10 揭示当把 new 运算符和异常一起使用时的内存泄漏问题

```
// destroy8.cpp
#include <cstdio>
using namespace std;

class File
{
    FILE* f;

public:
    File(const char* fname, const char* mode)
    {
        f = fopen(fname, mode);
        if (!f)
            throw 1;
    }
    ~File()
    {
        if (f)
        {
            fclose(f);
            puts("File closed");
        }
    }
};
```

```

main()
{
    void f(const char*);
    try
    {
        f("file1.dat");
    }
    catch(int x)
    {
        printf("Caught exception: %d\n",x);
    }
}

void f(const char* fname)
{
    File* xp = new File(fname,"r");
    puts("Processing file...");
    throw 2;
    delete xp;    //不会发生
}

// 输出:
Processing file...
Caught exception: 2

```

就像在程序清单 13.11 中看到的那样，可以在 `auto_ptr` 对象中“封装”一个 `new` 表达式的结果。这个 `auto_ptr` 对象的 `operator*` 和 `operator->` 成员把各自的操作传递到基本的指针中，这样就可以正常地使用它了。由于 `auto_ptr` 对象本身就存储在堆栈中，所以当它的析构函数被调用时析构函数依次为它包含的 `T*` 调用 `delete` 操作。

回想一下当在自由存储单元中创建对象时，如

```
T* P=new T;
```

系统在初始化新对象之前为它调用 `operator new` 来分配内存空间。如果 `T` 抛出异常，不必关心内存泄漏——运行期系统调用 `operator delete` 来释放 `operator new` 所分配的内存空间。C++ 总是在异常发生时清除部分建立的对象。

程序清单 13.11 用 `auto_ptr` 处理内存泄漏

```

// destroy9.cpp
#include <cstdio>
#include <memory>    //为使用自动指针
using namespace std;

class File
{
    FILE* f;

```

```

public:
    File(const char* fname, const char* mode)
    {
        f = fopen(fname, mode);
        if (!f)
            throw 1;
    }
    ~File()
    {
        if (f)
        {
            fclose(f);
            puts("File closed");
        }
    }
};

main()
{
    void f(const char*);
    try
    {
        f("file1.dat");
    }
    catch(int x)
    {
        printf("Caught exception: %d\n", x);
    }
}

void f(const char* fname)
{
    auto_ptr<File> xp = new File(fname, "r");
    puts("Processing file...");
    throw 2;
}

// 输出:
Processing file...
File closed
Caught exception: 2

```

13.7 内存管理

在异常这个概念出现之前,如果 new 操作分配内存失败则返回空指针,就像 C 中的 malloc

一样:

```
T*tp=new T;
If (tp)
    //使用新的对象
```

C++ 现在规定内存分配失败将导致 `bad_alloc` 异常。标准库和第三方的库大量使用自由存储。由于内存分配请求可能在最不希望它发生的时候出现, 几乎你的所有程序都应该有处理 `bad_alloc` 异常的准备。很显然办法是提供一个处理程序:

```
#include<new>
catch(const bad_alloc& x)
{
    cerr<<"out of memory: "<< x.what()<<endl;
    abort();
}
```

根据自己的应用程序可以做比这个处理程序更有意义的事, 如恢复内存并返回到稳定状态来重试。

如果喜欢返回空指针这种经典的做法, 可以用 `new` 操作符的特殊版本:

```
#include<new>
//...
T* tp=new(nothrow) T;
If (tp)
    //用 tp...
```

另一种处理内存溢出的方法是替换机器本身的部分内存分配机制。当内存分配失败时, C++ 调用默认的新处理程序, 这个处理程序抛出 `bad_alloc` 异常。可以通过向 `set_new_handler` 传递处理程序的地址来提供自己的处理程序, 这很像我在上面对 `set_terminate` 所做的那样。

13.8 异常规范

可以用异常规范来列举出函数抛出的异常:

```
class A;
class B;
void f( ) throw(A,B)
{
    //无论什么
}
```

这个定义表明只有 `A` 或 `B` 类型的异常可以脱离 `f`。除了作为好的文件, 运行期系统只检测那些异常发生时允许的类型是否直接在 `f` 中或是间接地在调用链的深处。在任何其他异常之前, 控制传递给标准的库函数 `unexpected`, 它在默认状态下终止程序。上面 `f` 的定义等价于:

```
void f( )
{
    try
    {
```

```

        //无论什么
    }
    catch(const A&)
    {
        throw; //重新抛出
    }
    catch(const B&)
    {
        throw; //重新抛出
    }
    catch(...)
    {
        unexpected;
    }
}

```

你可以提供自己的异常处理程序，方法是通过把这个异常处理程序传递给标准库函数 `set_unexpected`，如下所示：

```

// unexpected.cpp
#include <iostream>
#include<exception>
#include<cstdlib>
using namespace std;
main( )
{
    void f( ) throw( );
    void handler( );
    set_unexpected(handler);
    f( );
}
void f( ) throw( )
{
    throw 1;
}
void handler( )
{
    cout<<"Deleted unexpected exception\n";
    exit(EXIT_FAILURE);
}

```

//输出:

Deleted unexpected exception

定义 `void f() throw() {...}` 不允许任何从 `f` 中脱离的异常，就是说，它与下面的语句等价：

```

void f( )
{
    try
    {

```

```

        //无论什么
    }
    catch(...)
    {
        unexpected( )
    }
}

```

一个没有异常规范的函数可以抛出任何异常。

对异常规范的挑战是 f 可能调用抛出其他异常的其他函数。必须知道什么异常是可能发生的，然后或者在 f 的规范中包含它们，或者在 f 内部显式地处理它们。如果将来版本的服务函数 f 使用了新的异常，就成为一个问题。如果这些新的异常是从 A 或 B 派生来的，就不会有麻烦，但是这种情况并不可能发生在商用库中。好的规则是：如果 f 函数调用其他没有关联异常规范的函数，那么不用为 f 再声明一个规范。同时，由于模板参数经常可以是任何的类型，因此就不能预知哪个类型的成员函数将会发生什么异常。底线：模板和异常规范不要混合使用。

13.9 错误处理策略

还有一种错误我没有讨论，我喜欢称之为“我的愚蠢错误”。这里用第一人称的形式是重要的。无论你是否知道（但我希望你），所有的开发者在构造软件时都做逻辑假设。比如，很多情况下我可以说：“这个指针在这里不能是空的”，并且我知道如果我已经按照所计划的方法来精心地编程，那么就能保证这个条件总是满足的，为了使断言明显并且可实施，我加入了下面的语句：

```
assert (p);
```

如果当表达式括号中的值是 0，则这个在 `<assert.h>` 中定义的 `assert` 宏将会使程序终止，并且还会给出一个错误信息以说明它包含错误出现在行数和文件名。当我用 `assert` 宏时，就意味着我已经控制了管理 `assert` 表达式的所有情况。其他的开发者喜欢用 `assert` 检查用户错误，甚至用来检查参数的先置条件，但我认为这样是错的。断言只是断言，如果我没有已经控制了断言我就不会使用它。

用一个很棘手的例子来说明问题。从一个把对象的数据记录写入数据库的面向对象的永久框架来考虑下面的函数：

```

bool Update::write( )
{
    cRecordset *pRS=m_pTableX->GetRecordset( );
    assert(pRS);
    if (!pRS->CanUpdate( ))
        Throw2(PFX , UPDATE_ERROR, "Database not updatable");
    bool status;
    try

```

```

{
    status= pRS-> Update( );
}
catch (CDBException *ep)
{
    string msg=ep->m_strError + ep->m_strStateNativeOrigin;
    ep->Delete( );
    Th2row(PFX , UPDATE_ERROR,msg);
}
return status;
}

```

函数 `GetRecordset` 返回一个指向 `Recordset` 的指针，这个 `Recordset` 是用来做关系数据库 I/O 的 MFC 库抽象。我设计了该系统，保证 `PRS` 不会为空。如果 `PRS` 为空，那么我的软件就会崩溃。另一方面，我无法控制你所提供的数据库的连接是否是可更新的。所以我便抛出了异常（我将简短地解释 `Throw2` 宏）。这给了你，我的客户，一个在运行期处理事情的机会，如果你打算继续向前，而不是不得不退出程序并且重新开始。我认为它与向函数中输入参数的方法相同。所以我对错误处理的首先有以下几点建议：

1. 大胆地使用断言，但只能对已经直接控制的事情使用。
2. 为不能就地处理的错误抛出异常，包括无效的参数。

注意在上面的例子中，我用抛出自己异常标记的方法来处理数据库错误。那是因为我的客户不需要知道我用来访问数据的基本机制。他们所要知道的只是用我的组件以及我的组件可以抛出异常。那么他们不得不做的事就是要捕捉我所抛出的异常，否则标准 C++ 库将会捕捉所抛出的异常（例如我没有捕捉的内存失败），如下：

```

try
{
    //在这里调用我的一个函数
    p->Write( );
}
catch (PFX_Exception& x)
{
    //做你要做的，这里我只是打印一个消息：
    cout<<" PFX exception: "<<x.what( )<<endl;
}
catch(...)
{
    cout<<" Unknown exception"<< endl;
}

```

我固定使用的组件有许多类，但它只有一个从 `exception` 类中派生出来的 `exceptior` 类（见程序清单 13.12 和 13.13）。一个组件中有几个 `exception` 类，对客户来说只会使事情变得复杂。另外，我把我所有的异常都包括在它们的 `what` 信息中，包括异常抛出的地点的文件名和行数，因为这些重要信息用别的方法无法得到。为了这样做，我用到了下面的预处理技巧：

//xcept.h: 有用的抛出宏

```

#include<exception>
#include<string>
//为引证一个数字而使用的宏技巧
#define str_(x) #x
#define xstr_(x) str_(x)
//为方便异常抛出而使用的宏
//(包含标准信息、文件、行#)
#define Throw(Type,cod)\
throw Type ## _ Exception(Type ## _ Exception :: ## cod, \
std :: string(__FILE__ ## ":Line" \
## xstr_(__LINE__)))

```

程序清单 13.12 PFX_Exception 类头文件

```

// pfxexcept.h
#include "xcept.h"           //参见章节文本
using std::exception;
using std::string;

class PFX_Exception : public exception
{
public:
    PFX_Exception(int cod, const string& msg = "")
        : exception(s_ErrorStrings[cod] + ":" + msg)
    {}
    enum {BAD_OBJ_ID, CONNECT_ERROR, RECORDSET_ERROR,
        REQUERY_ERROR, UPDATE_ERROR, LOCK_ERROR,
        VERSION_ERROR, TRANS_ERROR, READ_ONLY_ERROR,
        INIFILE_ERROR,
        NUM_ERRORS};

private:
    static string s_ErrorStrings[NUM_ERRORS];
};

```

程序清单 13.13 PFX_Exception 的实现

```

// pfxexcept.cpp
#include "pfxexcept.h"

string PFX_Exception::s_ErrorStrings[NUM_ERRORS] =
{
    "Bad Objid",
    "Connection open failed",
    "Recordset open failed",
    "Recordset requery failed",
    "Recordset update failed",
    "Record lock failed",
    "Object out of date with database",
}

```



```

    "Transaction error",
    "Attempt to Write a ReadOnly object",
    "INI File name missing"
};

```

上面的抛出宏把调用，如：

```
Throw(PFX, LOCK_ERROR);
```

转变为语句：

```
throw PFX_ Exception(???);
```

这里???表示一个字符参数，它包含当前的文件名和行数，分别使用预定义宏 `_FILE_` 和 `_LINE_`。 `PFX_ Exception` 从 `exception` 派生来并支持根据 `exception` 构造函数的整数错误码（如 `LOCK_ERROR`）所预定义的字符串。上面的字符串化预处理操作符（`#`）在它的参数周围加上引号，并且标记传递符（`##`）把它的参数全并成一个单独的预处理符号。（这些操作符在第2章中已经讨论过）。 `Throw2` 允许在抛出点加入额外的文本。比如，如果捕捉到由下面表达式抛出的异常

```
Throw2(PFX, RECORDSET_ERROR, "Extra Text");
```

那么这个异常的 `what` 字符串将显示：

```
Recordset Open Error:Extra Text:txcept.cpp:Line 18
```

13.10 小结

可能关于异常要说的最重要的事情是只能在真正的异常环境中使用它们。像 `setjmp` 和 `longjmp`，异常干扰程序的正常控制。在一个典型的程序和函数的数量相比，应该有相对较少的异常处理程序。另外，异常只是为同步事件设计的，所以要意识到——异常与信号不要混合使用。出于有最高效率的出发点考虑，目前的经验表明由于 `try` 块及其相关嵌套函数的存在，异常使代码量增加 5%~15%。如果异常被抛出代码运行的速度会被影响。

- 不要在有构造函数和析构函数的对象前用 `setjmp` 和 `longjmp`。而是要用异常。当异常被抛出时，堆栈中所有的自动对象将被自动撤销。

- 异常允许独立的程序组件关于错误条件进行通信。
- 应该为程序中任何可能被抛出的异常提供处理程序。
- 通过与所抛出的对象的类型相匹配来选择处理程序。
- 根据被处理的类型从最特殊到最普通的顺序排列相邻的处理程序。
- 如果用标准库，准备好在标准继承结构里捕捉异常（至少是 `exception` 类型）。
- 在自动对象中封装资源分配以确保在需要的时候调用析构函数。
- 在 `auto_ptr` 中封装 `new` 表达式的结果以确保析构函数可以被合适地调用。
- 可以对文档使用异常规范并加强函数所抛出的异常类型。
- 断言只用来防止你（快速的开发者）的错误。

- 为每一个组件定义一个从 `exception` 类（或 `runtime_error` 或 `logic_error`，如果合适的话）派生出来的异常类。

面向对象编程

什么能使程序语言面向对象呢？曾经有一段时间由于市场的大肆宣传使得人们很难找到这个问题直接的答案。但答案是简单的：面向对象的语言支持三个概念：

1. 通过封装达到的数据抽象（见第7章）；
2. 继承；
3. 多态。

数据抽象，或定义新类型的能力，通过允许开发者给和手头上的问题相应的程序实体命名来增强程序的能力。例如人事应用程序可能会作为以下的超集定义一个 Employee 类：

```
class Employee
{
public:
    Employee(const string& ename,double erate) : name(ename)
    {
rate= erate;
    }
//访问接口:
string Getname( ) const {return name;}
double getRate( ) const {return rate;}
double getTimeWorked( ) const {return timeWorked;}

//方法:
void recordTime(double etime) { timeWorked=etime;}
double computePay( ) const;
private:
    string name;
    double rate;
```

```
        double timeWorked;
    };
double Employee:: computePay( ) const
{
    const double& hours= timeWorked;
    if (hours>40)
        return 40*rate+( hours-40)*rate*1.5;
    else
        return rate* hours;
}
```

有了这个类的定义就可以初始化 **Employee** 对象并调用它们的成员函数，例如：

```
main( )
{
    Employee e("John Hourly",16.50);
    e.recordTime(52.0);
    cout<<e.getName( )<< "gets"
        <<e. computePay( )<<endl;
}
```

//输出：

John Hourly gets 957.00

但如果有不同种类的雇员又会怎么样呢？例如，许多雇主认为钟点工和职员是有区别的。可以像下面这样定义 **SalariedEmployee** 类：

```
class SalariedEmployee
{
public:
    SalariedEmployee(const string& ename, double erate):name(ename)
    {
        rate=erate;
    }
    //访问方法：
    string Getname( ) const {return name;}
    double getRate ( ) const {return rate;}
    double getTimeWorked( ) const {return timeWorked;}
    //方法：
    void recordTime(double etime) { timeWorked = etime;}
    double computePay ( ) const;
private:
    string name;
    double rate;
    double timeWorked;
};
double SalariedEmployee:: computePay( ) const
{
    return rate * timeWorked;
}
```

这两个类的唯一差别是类名和 `computePay` 方法的实现。怎样才能利用共通性来节省键盘输入和代码空间呢？有经验的 C 语言开发者会增加一个 `type` 标记符，它是一个识别 `Employee` 特点的整型代码。用这种方法甚至不需要 `SalariedEmployee` 类。这样一个类的定义就可以像下面这样：

```
class Employee
{
public:
    Employee(const string &ename, double erate, int etype)
        : name( ename )
    {
        rate = erate;
        type=etype;
    }
    string  getName ( ) const { return name;}
    double  getRate( ) const { return rate;}
    double  getTimeWorked( ) const { return timeWorked;}

    void recordTime(double etime)  { timeWorked = etime;}
    double  computePay ( ) const;
    enum {HOURLY, SALARIED};

private:
    int type;
    string name;
    double rate;
    double timeWorked;
};

double Employee:: computePay ( ) const
{
    switch (type)
    {
    case HOURLY:
        if (timeWorked > 40)
            return 40*rate + (timeWorked - 40)*rate*1.5;
        else
            return rate* timeWorked;

    case SALARIED:
        return rate* timeWorked;
    }
}
```

这样的安排需要 `computePay` 测试对象的类型，这样它才能做正确的事情。用户现通过一个构造函数实参来显示他们想要的 `Employee` 对象的类型，如下：

```
main ( )
{
```

```

Employee e ("John Hourly ", 16.50, Employee::HOURLY);
e.recordTime (52.0);
cout << e. getName ( )<< "gets"
    <<e. computePay( )<<endl;
Employee e2("Jane Salaried",1125.00,Employee::SALARIED);
E2.recordTime(1.0);
Cout<<e2.getName()<<"gets"
    <<e2.computePay()<<endl;
}
//输出:

```

```

    John Hourly gets 957.00

```

```

    Jane Salaried gets 1125.00

```

现在, 如果需要一个新的雇员类型将会发生什么, **ExemptEmployee** 吗? 必须:

1. 在校举中增加新的类型标识符 **Exempt**, 并且
2. 在 **computePay** 的 **switch** 语句中增加一个 **case**。

这看起来工作量并不大, 但当依赖于雇员特点的方法的数量增加时, 它很快就成为维护程序的噩梦。这样做也失去在 **SalariedEmployee** 类中的优点: 能够使编译器和维护程序员清楚在解析空间中有两种截然不同的类型。那么开发者该怎么做呢?

14.1 继承

C++ 允许对 **SalariedEmployee** 按照如下 **Employee** 的第一版本进行声明:

```

//从 Employee 派生:
class SalariedEmployee : public Employee
{
public:
    SalariedEmployee (const string&, double);
    double computePay( ) const;
};
SalariedEmployee:: SalariedEmployee (const string& ename, double erate)
    : Employee ( ename, erate)
{ }
double SalariedEmployee:: computePay ( ) const
{
    return  getRate ( ) *getTimeWorked ( );
}

```

SalariedEmployee 类是 **Employee** 类的继承, 或是从 **Employee** 类中派生出来的。作为一个派生类, 它有 **Employee** 的一切, 也有权访问任何 **Employee** 的非私有成员。需要在派生类中定义的唯一的东西就是它区别于基类的内容——本例中是 **computePay** 的实现。**SalariedEmployee** 构造函数简单地把它被初始化的信息传递给在初始化列表中的 **Employee** 部分。现在便可以使用这两个类而不用亲自测试标志符, 并且没有任何重复的代码。更重要的是, 可以很简单地通过从 **Employee** 类 (或是从 **SalariedEmployee**, 如果合适的话) 中派生的

类来增加一个 `ExemptEmployee` 类，而不需要改变基类中的任何代码。

如果觉得用 `SalariedEmployee::computePay` 方式调用 `getRate` 和 `getTimeWorked` 很麻烦，可以在 `Employee` 中用 `protected` 代替 `private` 关键字：

```
protected:
    string name;
    double rate;
    double timeWorked;
```

保护(`protected`)成员可被所有派生对象使用，但在其他地方不能使用它们。可如下重写 `computePay`：

```
double SalariedEmployee:: computePay( ) const
{
    return rate * timeWorked;
}
```

正像通常定义数据成员为公有类型是个坏习惯一样，定义它们为保护类型也是个不好的倾向，因为派生的客户可能依赖于它们。最好的解决方法就是用保护成员函数去控制保护数据的访问。短语

```
class SalariedEmployee: public Employee
```

是说 `Employee` 是 `SalariedEmployee` 的公有基类。这就意味着非继承成员并不只能进行公有访问（这是普遍接受了的，这里不再累述）。换句话说，当通过派生类对象使用时基类成员没有设置限制。继承的类型在派生类和基类之间是一种“is-a”关系，因为由派生对象可访问的所有成员也是客户可以通过派生对象访问的。这是在 C++ 程序中最常见到的继承类型。使用保护(`protected`)继承，任何公有(`public`)基类成员在通过派生对象访问时都降为保护(`protected`)类型，所以任何派生类的客户都不能看到基类子对象的任何部分（只有派生对象可以）。任何私有(`private`)基类成员都不能通过派生对象以任何的方式被访问。要点：限定类派生的访问标识符是管理派生类对象客户访问权限的过滤器。然而，可以用访问声名来重载由于保护或私有继承所造成的任何降级。例如，在下面类的声明中，`B::f` 被恢复为 `D` 类客户的公有访问：

```
class B
{
public:
    void f();
    void g();
};
class D: protected B
{
public:
    B::f;    //保持 f 公有
};
```

当然，能给一个成员的访问权限不能超过它原来的访问权限。

14.2 不同种类的集合

C++ 允许一个基类的指针实际指向派生类对象，例如：

```
SalariedEmployee s("Another employee", 1000.00);
Employee *p = &s;
```

这与 `SalariedEmployee` 是一个 `Employee` 的事实一致，因此它能在任何 `Employee` 操作中代表一个 `Employee` 对象。这就允许单个数组中保存指向两种对象的指针，例如：

```
main()
{
    Employee e("John Hourly", 16.50);
    e.recordTime(52.0);
    SalariedEmployee e2("Jane Salaried", 1125.00);
    e2.recordTime(1.0);
    Employee*elist[] = { &e, &e2 };
    int nemp = sizeof elist / sizeof elist[0];
    for (int i=0; i<nemp; ++i)
        cout << elist[i]->getName() << "gets"
              << elist[i]->computePay() << endl;
}
```

然而上面的代码中隐藏着一个不易发现的错误在我给每个类中的 `computePay` 方法增加一条跟踪语句之后，它就出现了。

//输出：

```
Employee::computePay()
John Hourly gets 957.00
Employee::computePay()
Jane Salaried gets 1125.00
```

Jane 薪水的支付被错误的函数计算！它与正确答案得出几乎是同时发生的（这就是使这个错误不易被发现的原因）。由于 `elist` 被声明为指向 `Employee` 的指针数组，表达式 `elist[i]->computePay` 总是调用 `Employee::computePay`。缺乏任何其他的机制时，C++ 在编译期把函数调用与特定函数相绑定，这个过程被称做静态绑定。既然没有方法确定对象 `elist[i]` 实际指向哪一种类型，那么编译器就使用它自己拥有的唯一信息：指针类型（`Employee*`）。用户所要做的事情就是用某种方法告诉编译器将函数调用动态绑定到正确的类方法，这取决于对象 `elist[i]` 当时实际所指向的类型。

14.3 虚函数和多态

调用 `computePay` 动态绑定时所有需要做的就是 在 `Employee` 类里声明它是虚(virtual)的：

```
class Employee
{
```



```

    //...
    virtual double computePay( ) const;
    //...
};

```

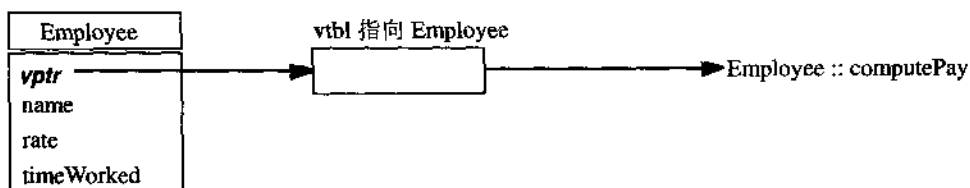


图 14.1 Employee 类的虚函数表

这就是它工作的原理图！现在上面程序的输出是：

```

//输出：
Employee::computePay()
John Hourly gets 957.00
Employee::computePay()
Jane Salaried gets 1125.00

```

这很容易。其作用是好像产生了一个运行期询问以确定调用对象类型，但是执行过程比这个原理图更加灵活有效。当一个类有一个或多个虚函数时，它有一个隐藏的指针数据成员，有时被称做 `vpitr`（读成“vee-pointer”）。这个隐藏的指针指向函数指针表，指针表中的元素依次指向该类所声明的虚函数（见图 14.1）。

当编译器看见对虚函数 `computePay` 的调用时，它通过当前对象的 `vpitr` 所指向的虚函数表产生一个间接的函数调用，有点像：

```
vpitr[0] //computePay 是 Employee 中的第 0 个虚函数
```

每个从 `Employee` 派生出来的类（直接或间接）都有它自己的虚函数表，并且每个派生对象的 `vpitr` 都指向那个 `vtbl`。因此，如果在派生类中重载任何继承的虚函数，它们会通过当前对象类型的虚函数表被间接地调用。对于在派生类中没有被重载的函数，相应的在派生虚函数表中的入口将从基类继承来的。在派生类中使用 `virtual` 关键字声明一个重载的继承虚函数是不必要的——一旦一个函数被声明为虚函数，那它在继承层次中将一直保持是虚函数。如果在派生类中增加新的虚函数，它们的地址会被附加到 `vtbl` 中。

练习 14.1

为了确信你理解了虚函数机制，在下面的程序中哪一个函数将被调用？哪一个函数不是虚函数？（答案在本章的最后。）

```

#include <iostream>
using namespace std;
class A
{

```

```
public:
virtual void f( )
virtual void g( )
class B : public A
{
public:
    void g( );
    virtual void h( )
};
class C: public B
{
public:
    void g( );
    void i( );
};

main ( )
{
    C c;

    A* pA = &c;
    pA->f();
    pA->g();

    B* pB = &c
    pB-> f();
    pB-> g();
    pB-> h();

    C* pC = &c
    pC-> f();
    pC-> g();
    pC-> h();
    pC-> i();
}
```

“多态”这个词源于希腊，它字面的意思是“多种形式”。在面向对象的思想中，它被用来将一个概念的多种表达和一个单一的名字联系起来，或换句话说，通过一个单独的接口传递变化的行为。虽然这听起来很复杂，但是要相信多态是一个很平常的概念。例如，考虑像汽车那样的交通工具（如，小汽车、卡车等等，见图 14.2）。你能驾驶一辆小汽车几乎就能驾驶任何一辆卡车，原因是它们的操控界面相同——方向盘、油门和刹车踏板，所有的都按标准方式安排。当你踩刹车踏板的时候，你正在发布停的命令，但是命令的具体内容可能是变化的，这取决于当时驾驶车辆的类型（例如大型卡车是典型的气动刹车，而不是盘刹车或鼓刹车，等等）。多态甚至对一个简单的数学表达式都起作用，如 $x+y$ 。如果这些符号出现在计算机的程序中，实际执行的微指令由对象究竟是整型还是浮点型指针数字决定。同样地，当

需要一个雇员的工资额去打印支票或报表时,并不想考虑它是如何计算的——只想得到结果。虚函数机制能使开发者为用户提供一个统一的接口,但是需要根据对象的动态类型自动地调整相应的行为,这只需在运行期对函数指针进行解引用便可实现。

14.4 抽象基类

有时一个类代表一个概念,例如一个完整的继承层次,它永远不会被实例化,如图 14.2 中的 `Vehicle` 层次。

在这种情况下 `Vehicle` 是一个组的机制。任何属于所有派生类的成员都属于 `Vehicle`,但不能实例化任何的 `Vehicle` 对象。这样的类被称为抽象类。在 C++ 中有两种方式来产生抽象类:

1. 至少声明一个纯虚函数;
2. 至少声明一个保护的构造函数,并且没有公有构造函数。

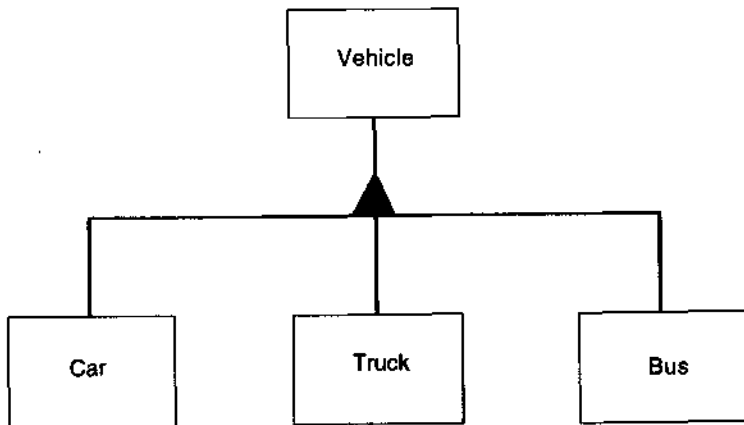


图 14.2 `Vehicle` 类层次

通过添加字符 `=0` 可以声明一个纯虚函数,如下:

```
virtual void stop() = 0; // vehicle::stop()
```

典型的,一个纯虚函数没有主体,因为期望每一个派生类都能实现自己的主体。纯虚函数在基类的集合决定了派生类必须实现的接口。编译器将禁止有纯虚函数的类创建对象。

已经讨论过的 `Employee` 类通过 `computePay` 实现的特点来支持钟点工。如果使用一个抽象类 `Employee` 可能会更有意义,它仅定义所有类型的雇员都共有的东西,如下所示:

```
class Employee
{
public:
    Employee(const string &ename, double erate) : name( ename )
    {
        rate = erate;
    }
};
```

```
    }

    string  getName ( ) const { return name;}
    double  getRate( ) const { return rate;}

    void  recordTime(double etime)  { timeWorked = etime;}
    virtual double  computePay ( ) const=0;

protected:
    string name;
    double rate;
    double timeWorked;
};
```

然后可以类似于 `SalariedEmployee` 定义 `HourlyEmployee`，像这样：

```
class HourlyEmployee : public Employee
{
public:
    HourlyEmployee(const string &ename, double erate)
: Employee( ename ,erate)
    { }
    double computePay( ) const;
};

double HourlyEmployee:: computePay( ) const
{
    cout<< "HourlyEmployee:: computePay( )\n";
    const double& hours = timeWorked;

    if (hours > 40)
        return 40*rate + (hours - 40)*rate*1.5;
    else
        return rate* hours;
}
```

抽象基类的另一个作用是赋予另一个类与其原来目的并不相关的一些功能。例如，在多用户的环境下，跟踪了解有多少客户正在共享一个特殊的资源是很关键的，所以不能过早地分配资源。下面的类实现参考计数：

```
class Counted
{
public:
    long Attach( ) {return ++m_RefCount;}
    long Detach( )
    {
        return ( --m_RefCount >0) ? m_RefCount
                                   : (delete this, 0);
    }
}
```

```

    long NumClients() const {return m_RefCount; }

Protected:
    Counted() { m_RefCount = 0; }
    Virtual ~Counted() { assert (m_RefCount == 0 );}
Private:
    Long m_RefCount;
};

```

因为惟一的构造函数是保护类型的，所以不能直接实例化 `Counted` 对象，但是 `Counted` 的派生类成员可以实例化 `Counted` 对象。`Attach` 方法增加计数，而 `Detach` 方法则减少计数。如果在 `Detach` 中，参考计数达到零，则对象通过 `delete` 运算符销毁它自己（所以资源一定已经通过 `new` 在堆栈中被大量创建）。使用这种类典型的习惯用法如下：

1. 在派生资源类中提供一个静态的 `Create` 方法，它用来在堆中分配一个新的派生类对象并且给它的 `Counted` 子对象产生一个初始的 `Attach`（增加 `count` 到 1）。这就保证了资源能被随后所有的客户使用。

2. 传递一个指针给资源作为客户构造函数的参数，它也能 `Attach` 资源。

3. 资源中的客户析构函数 `Detach`。

4. 当准备释放共享的资源时，做最后的 `Detach`。

下面的程序说明了这项技术：

```

Class Resource : public Counted
{
public:
    static Resource* Create()
    {
        Resource * pR = new Resource;
        pR->Attach();
        return pR;
    }
};

class Client
{
public:
    Client(Resource * pR)
    {
        pRes = pR;
        pRes->Attach();
    }
    ~Client() { pRes->Detach();}

private:
    Resource * pRes;
};

```

```

main ()
{
    //创建一个共享的资源:
    Resource * pR = Resource :: Create ( ); // count 是 1

    //使用资源:
    Client b1 {pR};                // count 是 2
    Client b2 {pR};                // cout 是 3

    //撤消初始化 Attach:
    pR->Detach ( );                //count 是 2
    //b2.Client: ~ Client()  将 count 减少到 1
    //b1.Client: ~ Client()  将 count 减少到 0
    //这些之后资源自动析构。
}

```

14.5 实例研究：一个对象持续的框架

面向对象范例的转化引起了对商业对象的关注。用户接口仅仅被看作是一种代表商业对象信息的方式，数据库存储一个对象的数据属性——越来越多的开发成果试图突出商业对象模型。商业对象自然而然地集中在一个或多个商业领域，例如销售或人员（见图 14.3）。

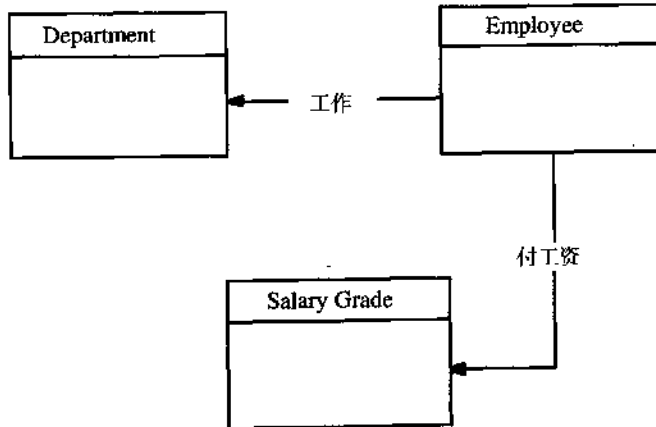


图 14.3 销售领域

面向对象编程语言为封装、继承和多态提供了机制，但是，像大部分其他的编程语言一样，不能直接地支持程序数据的永久存储。对这种缺点传统的解决方法是直接包含目录文件 I/O，使用嵌入式结构化查询语言(SQL)调用，或是依赖于第三方文件访问库。但是面向对象范例的关键特点是对对象应知道如何处理自身，客户简单地通过接口提出要求而并不关心细节的实现过程。例如，想给某个雇员增加薪水，在 C++ 中你可能希望能做如下事情：

```

//如果雇员# 23437 属于销售部，就给他增加薪水:

```

```

Employee emp(23437) ;
Department* dept = emp.GetDept ( ) ;
If (dept->GetType ( ) == SALES)
{
    emp.SetSalary(emp.GetSalary( ) * 1.25);
    emp.Write( );
}

```

这里不用担心有关文件名、域名，或是其他人工存储——对象知道该怎么做。

当然也不必总是知道对象的数字 ID 号，因此根据一些合理的标准应该可以重新得到对象清单，然后把它们中所有的或的一部分返回。下面的例子处理了一个雇员列表：

```

// 给销售部每个人增加薪水：
EmployeeQuery q;
q.AddSelect (OBJID, SALES);
q.Execute( );
while (!q.IsEof( ) )
{
    //实例化对象以调用它的方法：
    Employee emp(q.GetObjID( ));
    emp.SetSalary(emp.GetSalary( ) * 1.25);
    emp.Write( );
    q.Next( );    // 进行下一个记录
}

```

虽然提供方便功能的面向对象的数据库已经存在，它们仍然有些不够成熟，还未在共同的工作场所广泛的使用。本章其余的部分提出一个框架，以 PFX（即 Persistence Framework）命名，它给出了带有关系数据库的商业对象模型的持续性。在西方的美国大公司中 PFX 是实际产品的组成部分，它作为客户机/服务器结构的基本组件使用。

框架是一个紧密联系在一起的类集合，作为一个整体提供一些服务，如图形用户界面库。为了使用这个框架，习惯上要派生关键的框架类（通常是抽象的）以使它们的功能适应你的需求。定制框架通常是由重载框架提供的虚函数组成。PFX 是一个传统的由四个类组成的框架，其中的三个是抽象的，并在一个单独的库中打包（见图 14.4）。在面向对象的系统里框架是为复用而制定的主要传递机制：可以复用设计和所提供的部分实现，然后通过增加或替换部分实现来根据你的情况调整功能。另一个在 C++ 中复用的主要工具当然是模板设备，当没有必要定制功能时，但想把现有的服务应用到新类型上时可以应用它。

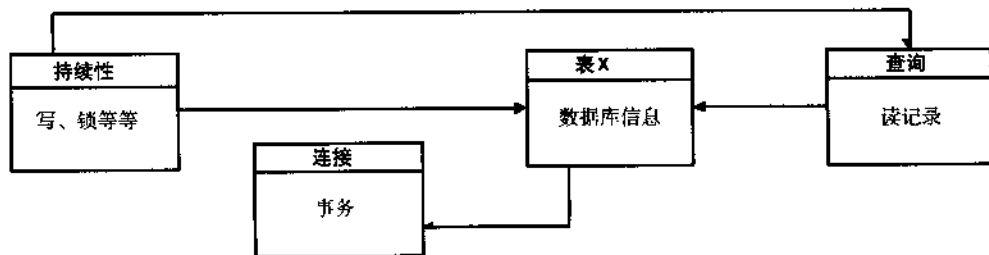


图 14.4 PFX 类

14.6 数据库访问

AFX 框架需要一些方式访问存放对象属性的数据库。今天，在客户/服务器环境中应用最为广泛的访问标准之一是微软公司的开放性数据库连接标准(ODBC)。ODBC 在 C 语言中定义了一个应用编程接口(API)标准来完成数据库处理和查询结果集。ODBC 驱动程序为最流行的数据库管理系统而存在。微软基本类库(MFC)被大多数以 Windows 为基础的 C++ 编译器所支持，它有两个类，CDatabase 和 CRecordset，它们把数据库请求翻译成 ODBC 调用。CRecordset 是一个抽象类，你可以把它作为基类派生新类完成对数据库表中数据的读写。例如，假设 Employee 表有下面的模式和数据：

OBJID	ENAME	JOB	MGID	HIREDATE	SALARY	DEPTID
7369	SMITH	CLERK	7902	17-DEC-80	800	20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	30
7566	JONES	MANAGER	7839	02-APR-81	2975	20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	30
7698	BLAKE	MANAGER	7839	01-MAY-81	2850	30
7782	CLARK	MANAGER	7839	09-JUN-81	2450	10
7788	SCOTT	ANALYST	7566	09-DEC-82	3000	20
7839	KING	PRESIDENT		17-NOV-81	5000	10
7844	TURNER	SALESMAN	7598	08-SEP-81	1500	30
7876	ADAMS	CLERK	7788	12-JAN-83	1100	20
7900	JAMES	CLERK	7698	03-DEC-81	950	30
7902	FORD	ANALYST	7566	03-DEC-81	3000	20
7934	MILLER	CLERK	7782	23-JAN-82	1300	10

在微软 Visual C++ 编译器中的类向导(Class Wizard)可以通过 ODBC 连接读取这个表，并且产生一个从 CRecordset 派生的记录集类(见程序清单 14.1 和 14.2)。可以用下面的(EmpRec)类：

```
#include "empr.h"
CDatabase db;
db.open("ODBC;DSN=Personnel");           // ODBC 连接字符串

EmpRec rec(&db);
rec.m_strFilter = " [Employee ID] = 5";    // WHERE 子句
rec.Open();
while (!rec.IsEOF())
{
    //处理记录，之后--
    rec.MoveNext();
}
```


程序清单 14.1 EmpRec 结果集接口

```
// EmpR.h : 头文件
#include <afxdb.h>    // 微软公司文件

class EmpRec : public Crecordset
{
public:
    EmpRec(CDatabase* pDatabase = NULL);

    // 字段/参数数据
    //{AFX_FIELD(EmpRec, CRecordset)
    double    m_ObjID;
    double    m_DeptID;
    CString   m_Name;
    CTime     m_HireDate;
    CString   m_Job;
    double    m_MgrID;
    double    m_Salary;
    //}}AFX_FIELD

    // 重载
    // ClassWizard 生成虚函数重载
    //{AFX_VIRTUAL(EmpRec)
    virtual CString GetDefaultConnect(); // 默认连接字符串
    virtual CString GetDefaultSQL();    // 用于记录集的默认 SQL
    virtual void DoFieldExchange(CFieldExchange* pFX); // RFX 支持
    //}}AFX_VIRTUAL
};
```

程序清单 14.2 EmpRec 类的实现

```
// EmpR.cpp : 实现文件 (FXW)
#include "EmpR.h"

EmpRec::EmpRec(CDatabase* pdb)
    : CRecordset(pdb)
{
    //{AFX_FIELD_INIT(EmpRec)
    m_ObjID = 0.0;
    m_DeptID = 0.0;
    m_Name = _T("");
    m_Job = _T("");
    m_MgrID = 0.0;
    m_Salary = 0.0;
    m_nFields = 7;
    //}}AFX_FIELD_INIT
```

```

        m_nDefaultType = snapshot;
    }

    CString EmpRec::GetDefaultConnect()
    {
        return _T("");
    }

    CString EmpRec::GetDefaultSQL()
    {
        return _T(" [EMPLOYEE] ");
    }

    void EmpRec::DoFieldExchange(CFieldExchange* pFX)
    {
        //{{AFX_FIELD_MAP(EmpRec)
        pFX->SetFieldType(CFieldExchange::outputColumn);
        RFX_Double(pFX, _T(" [OBJID] "), m_ObjID);
        RFX_Double(pFX, _T(" [DEPTID] "), m_DeptID);
        RFX_Text(pFX, _T(" [ENAME] "), m_Name);
        RFX_Date(pFX, _T(" [HIREDATE] "), m_HireDate);
        RFX_Text(pFX, _T(" [JOB] "), m_Job);
        RFX_Double(pFX, _T(" [MGRID] "), m_MgrID);
        RFX_Double(pFX, _T(" [SALARY] "), m_Salary);
        //}}AFX_FIELD_MAP
    }

```

记录集能处理任何表，也可用于数据库查询的只读操作中。

14.7 映射对象到相关模式

在关系数据库中储存对象的主要规则是“一个类对应一个表，一个类对象对应一行”。因为所有的对象必须有一个唯一的标识，出于这个目的它通常设有一个数值型字段，而且这个字段除了代表对象外没有别的作用。(在 PEX 中为对象标识可能可以使用其他方法，但是它需要用户做更多的工作)。然而有对象的包含，情况可能更复杂一点。如果每一个类对象 A 包含一个类型 B 的对象，在数据库中应该如何来反映呢？PEX 使用引用适应包含，这意味着，如果 B 是第一类商业对象——即和一个以数值为基础的对象相反，B 有对象标识例如一个日期或一个地址——那么应该在 A 表中有一个代表在 B 表中的一行的外部关键字。然后可以给出处理与 A 相关联的 B 对象的方法。如下：

```

class A
{
public:
    B* GetB () const {return new B(m_BobjID); }

```

```

Date GetBirth() const {return m_BiethDate; }
//.....
Private:
    //A的属性,包括:
    double m_BobjID;    //和对象B有关的外部关键字
    Date m_BirthDate;   //存储的值对象
};

```

其中“Get B”方法只有当它被调用时才提供它所包含的对象,这就节省初始化对象 A 的时间。PFX 中默认 Write 方法只存储立即对象(例如,只存储对象 A 的属性,而不是与 A 有关的对象 B 的属性),但是如果愿意的话也可以重载。

派生对象会是怎么样的呢? PFX 遵从“一个继承,一个表”的方法。在同一层次中的所有对象都共享相同的表,并且具有一个数值字段来标识每个对象的类型。就像在 Oracle DBMS 中一样,有了可变长度的存储结构,就不用担心当在层次中存储不使用所有属性的对象时由于容量增加而造成的空间浪费问题。除此之外,在一个定义明确的类层次中,大多数变量趋向于在行为中,而不是在属性中。

14.8 PFX 的结构

PXF 由 4 个主要的类组成,其中 3 个是抽象类并且必须由用户具体定义(如图 14.5 所示)。3 个将被具体定义的类如下:

1. 表 X (Table X) 提供对表信息的访问,和基本的数据库操作,如锁定记录、删除记录、检查记录是否存在以及在表中找出下一个可用的与商业对象相关联的数字作为一个对象的 ID。用户派生类必须给出表名和表中所有字段的名字。
2. 查询(Query) 提供办法以阐述查询标准和导航查询结果集。用户类提供访问特殊属性的方法。
3. 持续性为创建对象和把它们写到数据库中提供基本的方法。用户要给出读取和修改属性的方法,还要根据需要添加商业函数。

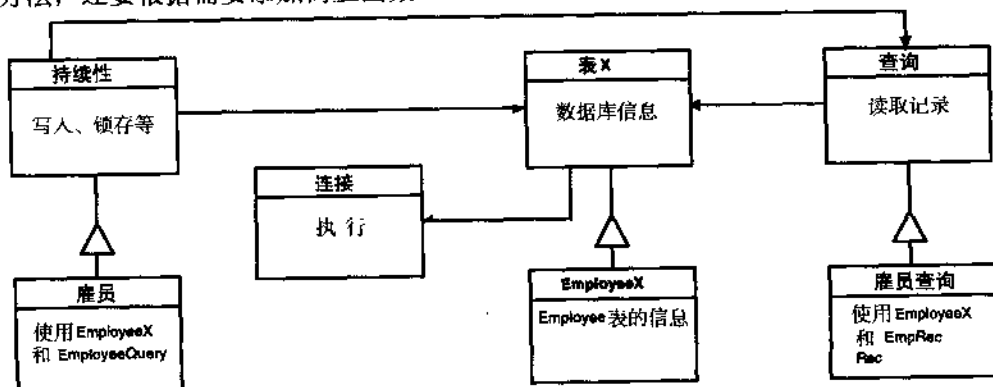


图 14.5 使用 PFX

第四种 PXF 类 **Connection** 是一个具体类，它封装了数据库的连接行为，包括事务处理。因为连接是一个共享资源，它是由 **Counted** 类派生出来的：

```
//connect.h :连接类接口
class Connection :public CDatabase,public Counted
{
    typedef CDatabase Super;

public :
    static Connection *Create(const CString&,BOOL rOnly=FLASE);
    Connection*Copy(BOOL rOnly=FLASE) const;
    CString GetConnectionStr() const;
    Void BeginTrans();
    Void Commit();
    Void Rollback();
    BOOL InTrans() const;

protected:
    Connection (const CString& conSpec,BOOL readOnly=FLASE);
    Virtual ~Connection();
    BOOL m_InTransFlag;          //当前是否在执行里?
};
```

Connection 类通过调用 **CDatabase** 类中的相应成员函数来做它的大部分工作(见程序清单 14.3)。所有的 PXF 类都有保护构造函数：**Connection**，因为它必须存储在堆中（而且只有通过它的静态 **Create** 函数生成）；其余的三个，因为它们是抽象类，所以必须具体化。

程序清单 14.3 Connection 类的实现

```
// connect.cpp
#include "connect.h"
#include "pfxxcept.h"

using namespace Persistence;

Connection* Connection::Create(const CString& conStr, BOOL rOnly)
{
    Connection* pCon = new Connection(conStr,rOnly);
    pCon->Attach();
    return pCon;
}

Connection* Connection::Copy(BOOL rOnly) const
{
    Connection* pNewCon = new Connection(GetConnectionStr(),rOnly);
    pNewCon->Attach();
    return pNewCon;
}
```

```

Connection::Connection(const CString& conSpec, BOOL rOnly)
{
    //打开连接
    try
    {
        int options = CDatabase::noOdbcDialog |
                      CDatabase::useCursorLib;
        if (rOnly)
            options |= CDatabase::openReadOnly;
        OpenEx(conSpec, options);

        //将 AUTOCOMMIT 关闭.
        ASSERT(m_hdbc);
        RETCODE result = SQLSetConnectOption(m_hdbc,
                                              SQL_AUTOCOMMIT,
                                              SQL_AUTOCOMMIT_OFF);

        if (result == SQL_ERROR)
            Throw2(PFX, CONNECT_ERROR,
                  _T("Set AUTOCOMMIT off failed"));
    }
    catch (CDBException* ep)
    {
        CString msg = ep->m_strError + ep->m_strStateNativeOrigin;
        ep->Delete();
        Throw2(PFX, CONNECT_ERROR, msg);
    }
    m_InTransFlag = FALSE;
}

void Connection::BeginTrans()
{
    if (!CanTransact())
        Throw(PFX, TRANS_ERROR);

    //开始重复调用
    if (!m_InTransFlag)
    {
        Super::BeginTrans();
        m_InTransFlag = TRUE;
    }
}

void Connection::Commit()
{

```

```

        if (!CanTransact() || !m_InTransFlag)
            Throw(PFX, TRANS_ERROR);

        Super::CommitTrans();
        m_InTransFlag = FALSE;
    }

void Connection::Rollback()
{
    if (!CanTransact() || !m_InTransFlag)
        Throw(PFX, TRANS_ERROR);

    Super::Rollback();
    m_InTransFlag = FALSE;
}

```

14.9 一个代码的预排

既然 PFX 使用 MFC，那么它不得不遵从 MFC 的规定，其规定包括：

- 用 MFC 的 CString 类代替 std::string；
- 用 MFC 的 BOOL 类型代替 bool，此时 FLASE/TRUE 代替 flase/true；
- 为了字符集的可移植性，所有引用的字符串都封装在-T 宏中。

当我们回顾 PFX 代码时要记住这些。

从 PFX 框架中继承的过程是很机械的，很容易自动化。实际上在这章中，读取记录集类定义的专用工具产生所有的文件。（就是程序清单 14.4、14.6、14.8、14.11 和 14.12）。

程序清单 14.4 包含数据字典，它是一个含有数据库信息的文本文件，且是标准 Windows 初始化文件格式的。程序清单 14.5 有 TableX 类接口，因为客户必须在派生类中提供实际的信息，所以 TableX 类把与数据库有关的函数定义成纯虚函数。它也实现函数来锁定或删除记录，并且获得下一个可用对象的 ID 数字。程序清单 14.6 的 EmployeeX 类为在构建查询中使用的字段名定义数值代码。

程序清单 14.7 中定义的 Query 类有查询构建和查询导航函数。为了防止用户想在查询被送到数据库执行之前做特殊的处理，它的执行函数是虚函数。在程序清单 14.8 中，EmployeeQuery 通过属性名提供访问函数从而获得查询结果。由于所有的 EmployeeQuery 成员是很小的，也很容易嵌入，所以不需要 EmpQ.cpp 文件。

程序清单 14.4 数据字典

// Test.ini:本项目包含的数据库信息

```

[Connection]
DEBUG=DSN=Oracle7-tables;UID=scott;PWD=tiger

```

```
RELEASE=DSN=Oracle7-tables;UID=scott;PWD=tiger
```

```
[Department]
Table=DEPARTMENT
SequenceStr=S_Department
NumFields=3
0=OBJID
1=LOCATION
2=NAME
```

```
[Employee]
Table=EMPLOYEE
SequenceStr=S_Employee
NumFields=7
0=OBJID
1=DEPTID
2=ENAME
3=HIREDATE
4=JOB
5=MGRID
6=SALARY
```

程序清单 14.5 TableX 类的接口

```
// tableX.h
class TableX
{
public:
    virtual ~TableX();
    virtual CString BusObjName() const = 0;
    virtual CString TableName() const = 0;
    virtual CString SequenceStr() const = 0;
    virtual int NumFields() const = 0;
    virtual CString GetIniFile() const;

    CString ConnectionStr() const;
    Connection* GetConnection() const;
    CString SQLName(int fldNum) const;
    double NextVal() const;
    BOOL Exists(double objID) const;
    void Lock(double id) const;
    void Remove(double id) const;
    CString GetProfileValue(const CString& category,
                           const CString& key) const;

    static CString NumericVal(double);
```

```
static CString StringVal(const CString&);
static CString StringVal(double);
static CString DateVal(const FullDate& d);
static void SetIniFile(const CString& fname);
static double NextVal(const CString& seq, Connection* pCon);
static CString& PrepareDBString(CString& s);
static void Remove(Connection* pCon, const CString& table,
                    const CString& idName, double idVal);

enum {NUMERIC_NULL = -1};

protected:
    TableX(Connection* pCon);
    Connection* m_pConnection;
    static CString s_IniFile;
};
```

程序清单 14.6 EmployeeX 类的接口

```
// EmpX.h ——包含 EmployeeX 声明
#include "tablex.h"

class EmployeeX : public TableX
{
    typedef TableX Super;

public:
    // 字段标识
    enum
    {
        OBJ_ID,
        DEPT_ID,
        NAME,
        HIRE_DATE,
        JOB,
        MGR_ID,
        SALARY,
        NUM_IDS
    };

    EmployeeX(Connection* pCon);

    // 重载
    CString TableName() const;
    CString BusObjName() const;
```



```

        CString SequenceStr() const;
        int NumFields() const;

//部门代码
        enum {ACCOUNTING = 10, RESEARCH = 20, SALES = 30,
              OPERATIONS = 40};

        private:
            static const CString s_BusObjName;
            static CString s_TableName;
            static CString s_SequenceStr;
            static CStringArray s_FieldNames;
            static int s_NumFields;
            static BOOL s_LoadedFlag;

//用来装载字段名的函数重载
            CStringArray& FieldList();
            BOOL FieldsLoaded() const;
            void MarkFieldsLoaded();
    };

    inline EmployeeX::EmployeeX(Connection* pCon)
        : TableX(pCon)
    {}

```

程序清单 14.7 Query 类的接口

```

// query.h
class Query
{
public:
    ~Query();
    virtual BOOL Execute(UINT = CRecordset::snapshot);
    int NumFields() const;

//查询导航函数
    BOOL First();
    BOOL Last();
    BOOL Next();
    BOOL Prev();
    // (为节省空间而省略的函数)

// 查询构造函数
    // (为节省空间而省略的函数)

//排序函数

```

```
// (为节省空间而省略的函数)

protected:
    Query(TableX*, CRecordset*);

    CRecordset* m_pRecordset;
    TableX* m_pTableX;
    CString m_Filter;
    CString m_Sort;
};
```

程序清单 14.8 EmployeeQuery 类的接口和实现

```
// EmpQ.h —— 包含 EmployeeQuery 的声明
#include <afxdb.h>           //为了 CRecordset 常数和 CString
#include "query.h"
#include "fulldate.h"
#include "empx.h"
#include "empr.h"

class EmployeeQuery : public Query
{
    typedef Query Super;
    typedef EmpRec* RecPtr;      //映射到正确的 Recordset 类型
public:
    EmployeeQuery(Connection* pCon);

    //为 Employee 附加的存取器
    double GetObjID() const;
    double GetDeptID() const;
    CString GetName() const;
    LDS_Date::FullDate GetHireDate() const;
    CString GetJob() const;
    double GetMgrID() const;
    double GetSalary() const;
};

inline EmployeeQuery::EmployeeQuery(Connection* pCon)
    : Query(new EmployeeX(pCon), new EmpRec(pCon))
{}

inline double EmployeeQuery::GetObjID() const
{
    using TableX;
    return ID(CLIENT_REC->m_ObjID);    //用于标识的特殊 TableX 宏
```

```

}

inline double EmployeeQuery::GetDeptID() const
{
    double temp = CLIENT_REC->m_DeptID;
    return (temp == AFX_RFX_DOUBLE_PSEUDO_NULL) ? TableX::NUMERIC_NULL
                                                : temp;
}

//(为节省空间而省略了其他的 Get 函数)

```

程序清单 14.9 Persist 类的接口

```

// persist.h
class Persist
{
public:
    virtual ~Persist();

//操作对象
    virtual void Write();
    virtual void WriteAsNew();
    virtual void Remove();
    virtual void Lock();
    void Refresh();

//存取器,等等
    double GetObjID() const;
    Connection* GetConnection() const;
    TableX* GetX() const;
    virtual BOOL IsDirty() const = 0;
    BOOL IsValid() const;
    operator void*() const;

protected:
    Persist(TableX* pX, double id = TableX::NUMERIC_NULL);
    virtual BOOL Read_(double objID) = 0;
    virtual CString InsertValuesStr() const = 0;
    virtual CString UpdateValuesStr() const = 0;
    virtual void ClearFieldFlags() = 0;
    virtual void MarkFieldFlags() = 0;
    double NextID() const;
    void AddRec();
    void UpdateRec();

//对全部固定对象是公有的属性:

```

```

    double m_ObjID;
    TableX* m_pTableX;
    BOOL m_ValidFlag;
};

```

Persist 类有读取和写入单一记录的方法(见程序清单 14.9 和 14.10)。有尽可能多的逻辑已经添加到了 Persist 类中,只把必须由派生类做的工作留给派生类去完成。Employee 类(见程序清单 14.11 和 14.12)是从 Persist 类中派生出来的,它为 Persist 类提供在插入和更新记录时使用的属性和字段名。Employee 中的最后的四个函数——WhoIWorkFor、WorWorksForMe、GetBoss 和 GetDept——已超出了简单的 I/O,它们仅仅由代码组成,这些代码不是由机器产生的(换句话说,这是最精彩的部分!)。头两个函数利用这个系统使用的是 Oracle 数据库这个实际情况,然后指定 SQL 代码。

程序清单 14.10 Persist 类的实现

```

// persist.cpp
#include "persist.h"
#include "pfxxcept.h"
#include "util.h"

using namespace Persistence;
using namespace Utility;

void Persist::Write()
{
    if (IsDirty())
    {
        Connection* pCon = GetConnection();

        //强制事务处理:
        if (pCon->CanTransact() && !pCon->InTrans())
            pCon->BeginTrans();

        if (m_ValidFlag)
            UpdateRec();
        Else
            AddRec();

        ASSERT(IsValid());
        ClearFieldFlags();
    }
}

void Persist::Remove()
{

```

```

        if (m_ObjID == TableX::NUMERIC_NULL || !m_ValidFlag)
            Throw(PFX, BAD_OBJ_ID);

//删除这个对象的记录:
    ASSERT(m_pTableX);
    m_pTableX->Remove(m_ObjID);
    m_ValidFlag = FALSE;
}

void Persist::Lock()
{
    if (m_ObjID <= 0 || !m_ValidFlag)
        Throw2(PFX, LOCK_ERROR, _T("No record in database"));

    m_pTableX->Lock(m_ObjID);
}

void Persist::Refresh()
{
    if (m_ObjID == TableX::NUMERIC_NULL || !Read_(m_ObjID))
        Throw(PFX, READ_ERROR);
    ClearFieldFlags();
    ASSERT(m_ValidFlag);
}

void Persist::AddRec()
{
    ASSERT(m_ObjID == TableX::NUMERIC_NULL);    //我可以这样做吗?

    Connection* pCon = GetConnection();
    if (!pCon->CanUpdate())
        Throw2(PFX, INSERT_ERROR, _T("Connection not updatable"));

    m_ObjID = NextID();
    ASSERT(m_ObjID);

    ASSERT(m_pTableX);
    CString sql = _T("insert into ") + m_pTableX->TableName()
        + _TCHAR(' ') + InsertValuesStr();

    try
    {
        pCon->ExecuteSQL(sql);
    }
    catch (CDBException* ep)
    {

```

```

        CString msg = ep->m_strError + ep->m_strStateNativeOrigin;
        ep->Delete();
        Throw2(PFX, INSERT_ERROR, msg);
    }

    m_ValidFlag = TRUE;
}

void Persist::UpdateRec()
{
    Connection* pCon = GetConnection();
    if (!pCon->CanUpdate())
        Throw2(PFX, UPDATE_ERROR, _T("Connection not updatable"));
    //如果没有脏字段将不做任何事:
    CString updateValues = UpdateValuesStr();
    if (updateValues.IsEmpty())
        return;

    ASSERT(m_ObjID > 0);

    ASSERT(m_pTableX);
    CString sql = _T("update ") + m_pTableX->TableName()
        + _T(" set ") + UpdateValuesStr()
        + _T(" where ") + m_pTableX->SQLName(0)
        + _T(" = ") + NumToStr(m_ObjID);

    try
    {
        pCon->ExecuteSQL(sql);
    }
    catch (CDBException* ep)
    {
        CString msg = ep->m_strError + ep->m_strStateNativeOrigin;
        ep->Delete();
        Throw2(PFX, UPDATE_ERROR, msg);
    }
}

```

程序清单 14.11 Employee 类的接口

```

// Emp.h——包含 Employee 声明
#include <afx.h>
#include "persist.h"
#include "fulldate.h"
#include "emp.h"
#include "bits.h"

```

```

class EmployeeQuery;
class Department;           //手动添加

class Employee : public Persist
{
public:
    //构造函数
    Employee(Connection* pClient,
               double id = TableX::NUMERIC_NULL);
    Employee(const EmployeeQuery& q);
    //存取器
    double GetDeptID() const;
    CString GetName() const;
    FullDate GetHireDate() const;
    CString GetJob() const;
    double GetMgrID() const;
    double GetSalary() const;
    BOOL IsDirty() const;

    //赋值
    void SetDeptID(double deptID);
    void SetName(const CString& name);
    void SetHireDate(const FullDate& hireDate);
    void SetJob(const CString& job);
    void SetMgrID(double mgrID);
    void SetSalary(double salary);

    //其他操作(手动添加)
    EmployeeQuery* Employee::WhoIWorkFor() const;
    EmployeeQuery* Employee::WhoWorksForMe() const;
    Employee* Employee::GetBoss() const;
    Department* Employee::GetDept() const;

    static BOOL Exists(Connection* pCon, double objid);

protected:
    BOOL Read_(double id);           //重载
    CString InsertValuesStr() const; //重载
    CString UpdateValuesStr() const; //重载
    void ClearFieldFlags();          //重载
    void MarkFieldFlags();           //重载
    TableX* MakeX() const;

private:

```

```
// 私有的数据成员
    double m_DeptID;
    CString m_Name;
    FullDate m_HireDate;
    CString m_Job;
    double m_MgrID;
    double m_Salary;
    bits<EmployeeX::NUM_IDS> fieldFlags;
//不能使用的操作(不被实现)
    Employee(const Employee&);
    void operator=(const Employee&);
};

inline double Employee::GetDeptID() const
{
    return m_DeptID;
}

// (保留为节省空间而省略的 Get 函数)
inline void Employee::SetDeptID(double deptID)
{
    m_DeptID = deptID;
    fieldFlags.set(EmployeeX::DEPT_ID);
}

// (保留为节省空间而省略的 Set 函数)
inline BOOL Employee::Exists(    Connection* pCon, double objid)
{
    return EmployeeX(pCon).Exists(objid);
}

inline void Employee::ClearFieldFlags()
{
    fieldFlags.reset();
}

inline void Employee::MarkFieldFlags()
{
    fieldFlags.set();
}

inline BOOL Employee::IsDirty() const
{
    return fieldFlags.any();
}
```


程序清单 14.12 Employee 类的实现

```

// Emp.cpp——处理 Employees 的类
#include "emp.h"
#include "empq.h"
#include "empx.h"
#include "connect.h"
#include "pfxxcept.h"
#include "dept.h"

Employee::Employee(Connection* pClient, double id)
    : Persist(new EmployeeX(pClient), id),
      m_Name(_T("")),
      m_HireDate(_T("")),
      m_Job(_T(""))
{
    if (!Read_(id))
    {
        m_DeptID = TableX::NUMERIC_NULL;
        m_MgrID = TableX::NUMERIC_NULL;
        m_Salary = TableX::NUMERIC_NULL;
    }
}

Employee::Employee(const EmployeeQuery& q)
    : Persist(new EmployeeX(q.GetConnection()), q.GetObjID())
{
    if (!q.IsEOF() && !q.IsBOF())
    {
        m_DeptID = q.GetDeptID();
        m_Name = q.GetName();
        m_HireDate = q.GetHireDate();
        m_Job = q.GetJob();
        m_MgrID = q.GetMgrID();
        m_Salary = q.GetSalary();
        m_ValidFlag = TRUE;
    }
}

BOOL Employee::Read_(double id)
{
    m_ValidFlag = FALSE;

    if (id > 0)
    {
        // 执行 objid 查询:
    }
}

```

```

        EmployeeQuery q(GetConnection());
        q.AddSelect(EmployeeX::OBJ_ID, id);
        q.Execute();
// 初始化数据成员 (m_pConnection 和 m_ObjID 已经被处理过)
        if (!q.IsEOF() && !q.IsBOF())
        {
            m_DeptID = q.GetDeptID();
            m_Name = q.GetName();
            m_HireDate = q.GetHireDate();
            m_Job = q.GetJob();
            m_MgrID = q.GetMgrID();
            m_Salary = q.GetSalary();
            m_ValidFlag = TRUE;
        }
    }

    return m_ValidFlag;
}

CString Employee::InsertValuesStr() const
{
// 构造 INSERT 词名的一部分:
    CString fields = _T(" ") + m_pTableX->SQLName(0);
    int nFields = m_pTableX->NumFields();
    for (int i = 1; i < nFields; ++i)
        fields += _T(", ") + m_pTableX->SQLName(i);
    fields += _T(") values ");

// 这个命令必须匹配 Registry ColumnName 命令!!!

// (这是由框架向导生成的)
    CString values = _T(" ") + TableX::NumericVal(m_ObjID);
    values += _T(", ") + TableX::NumericVal(m_DeptID);
    values += _T(", ") + TableX::StringVal(m_Name);
    values += _T(", ") + TableX::DateVal(m_HireDate);
    values += _T(", ") + TableX::StringVal(m_Job);
    values += _T(", ") + TableX::NumericVal(m_MgrID);
    values += _T(", ") + TableX::NumericVal(m_Salary);
    values += _T(")");

    return fields + values;
}

CString Employee::UpdateValuesStr() const
{

```

```

// 为脏字段组合 SET 子句:

// (循环开始在 1, 因为对象的标识字段为段 0, 并且一定不能被改变)
CString result;
for (int i = 1; i < EmployeeX::NUM_IDS; ++i)
{
    if (fieldFlags.test(i))
    {
        if (!result.IsEmpty())
            result += _TCHAR(',');
        result += m_pTableX->SQLName(i) + _T("=");

        switch (i)
        {
            case EmployeeX::DEPT_ID:
                result += TableX::NumericVal(m_DeptID);
                break;
            case EmployeeX::NAME:
                result += TableX::StringVal(m_Name);
                break;
            case EmployeeX::HIRE_DATE:
                result += TableX::StringVal(m_HireDate.ToString());
                break;
            case EmployeeX::JOB:
                result += TableX::StringVal(m_Job);
                break;
            case EmployeeX::MGR_ID:
                result += TableX::NumericVal(m_MgrID);
                break;
            case EmployeeX::SALARY:
                result += TableX::NumericVal(m_Salary);
                break;
        }
    }
}

return result;
}

// 这四种方法是手工代码:
EmployeeQuery* Employee::WhoIWorkFor() const
{
    EmployeeQuery* pQ = new EmployeeQuery(GetConnection());
    CString filter;
    filter.Format(_T("1 = 1 connect by objid = prior mgrid")

```

```

        " start with objid = %.0f"), m_ObjID);
    pQ->SetFilter(filter);
    pQ->Execute();
    return pQ;
}

EmployeeQuery* Employee::WhoWorksForMe() const
{
    EmployeeQuery* pQ = new EmployeeQuery(GetConnection());
    CString filter;
    filter.Format(_T("1 = 1 connect by prior objid = mgrid"
        " start with objid = %.0f"), m_ObjID);
    pQ->SetFilter(filter);
    pQ->Execute();
    return pQ;
}

Employee* Employee::GetBoss() const
{
    return m_MgrID ? new Employee(GetConnection(), m_MgrID) : 0;
}

Department* Employee::GetDept() const
{
    return m_DeptID ? new Department(GetConnection(), m_DeptID) : 0;
}

```

程序清单 14.13 说明 Employee 和 Department 对象持续性的程序

// test.cpp: 举例说明 Employee 和 Department 类

```

#include <iostream.h>
#include "connect.h"
#include "dept.h"
#include "deptQ.h"
#include "emp.h"
#include "empQ.h"
#include "tablex.h"
#include "util.h"
#include "rcdexcept.h"

using namespace Persistence;
using namespace Utility;

main()
{
    CString iniFile = _T("/test.ini");

```

```
TableX::SetIniFile(iniFile);
CString conStr = GetProfileValue(iniFile,_T("Connection"),
                                _T("Debug"));

Connection* pC;
Try
{
    cerr << "Connecting... " << endl;
    pC = Connection::Create(conStr);
    ASSERT(pC);
}
catch (Exception& x)
{
    cerr << x.GetMessage() << endl;
    cin.get();
    return 1;
}
cout << endl;

cout << "Departments: " << endl;
cout << "======" << endl;

DepartmentQuery q(pC);
q.Execute();
while (!q.IsEOF())
{
    cout << q.GetObjID() << ', '
         << q.GetName() << ', '
         << q.GetLocation() << endl;
    q.Next();
}
cout << endl;

cout << "Employees: " << endl;
cout << "======" << endl;

EmployeeQuery q2(pC);
q2.Execute();
while (!q2.IsEOF())
{
    cout << q2.GetObjID() << ', '
         << q2.GetName() << ', '
         << q2.GetHireDate().ToString() << ', '
         << q2.GetSalary() << ', '
         << q2.GetJob() << endl;
    q2.Next();
}
```

```
    cout << endl;
// 检查销售部雇员的数据:
    cout << "Sales Dept:\n";
    cout << "=====\n";
    q2.ResetSelect();
    q2.AddSelect(EmployeeX::DEPT_ID, EmployeeX::SALES);
    q2.Execute();
    double empID = 0;
    while (!q2.IsEOF())
    {
        CString name = q2.GetName();
        cout << name << endl;
        if (name == "TURNER")
            empID = q2.GetObjID(); //保存 TURNER 的 ID
        q2.Next();
    }
    cout << endl;

    if (empID)
    {
//用具体例证说明 TURNER 和他的老板:
        Employee e(pC, empID);
        Employee* pBoss = e.GetBoss();
        cout << "TURNER's Boss: ";
        cout << pBoss->GetName() << endl;

//实例化他的部门:
        Department* pDept = e.GetDept();
        cout << "TURNER's Department: "
            << pDept->GetName() << endl;
        delete pDept;
        cout << endl;

//打印 TURNER 的同事:
        EmployeeQuery* pQ = pBoss->WhoWorksForMe();
        cout << "Who works for TURNER's boss:\n";
        while (!pQ->IsEOF())
        {
            cout << pQ->GetName() << endl;
            pQ->Next();
        }
        delete pQ;
        delete pBoss;
        cout << endl;
    }
}
```

```

//打印以上 TURNER 行的命令:
    pQ = e.WhoIWorkFor();
    cout << "Who TURNER works for:\n";
    while (!pQ->IsEOF())
    {
        cout << pQ->GetName() << endl;
        pQ->Next();
    }
    delete pQ;
}

//创建一个新部门:
    Department d(pC);
    d.SetName("ICS");
    d.SetLocation("CHQ");
    d.Write();
    pC->Commit();
    double deptNo = d.GetObjID();

//增加新的雇员 :
    Employee e(pC);
    e.SetName("Einstein");
    e.SetDeptID(deptNo);
    e.SetHireDate(CString(_T("19970512")));
    e.SetJob(_T("Genius"));
    e.SetSalary(10000);
    e.Write();
    double mgrID = e.GetObjID();

    e.SetName("Feynman");
    e.SetDeptID(deptNo);
    e.SetHireDate(CString(_T("19970513")));
    e.SetJob(_T("Asst. "));
    e.SetSalary(5000);
    e.SetMgrID(mgrID);
    e.WriteAsNew();

    pC->Commit();
    pC->Detach();
    cin.get();
    return 0;
}

```

程序清单 14.14 程序清单 14.13 的输出

Departments:

=====

10, ACCOUNTING, NEW YORK

20, RESEARCH, DALLAS

30, SALES, CHICAGO

40, OPERATIONS, BOSTON

Employees:

=====

7369, SMITH, 19801217, 800, CLERK

7499, ALLEN, 19810220, 1600, SALESMAN

7521, WARD, 19810222, 1250, SALESMAN

7566, JONES, 19810402, 2975, MANAGER

7654, MARTIN, 19810928, 1250, SALESMAN

7698, BLAKE, 19810501, 2850, MANAGER

7782, CLARK, 19810609, 2450, MANAGER

7788, SCOTT, 19821209, 3000, ANALYST

7839, KING, 19811117, 5000, PRESIDENT

7844, TURNER, 19810908, 1500, SALESMAN

7876, ADAMS, 19830112, 1100, CLERK

7900, JAMES, 19811203, 950, CLERK

7902, FORD, 19811203, 3000, ANALYST

7934, MILLER, 19820123, 1300, CLERK

Sales Dept:

=====

ALLEN

WARD

MARTIN

BLAKE

TURNER

JAMES

TURNER's Boss: BLAKE

TURNER's Department: SALES

Who works for TURNER's boss:

BLAKE

ALLEN

WARD

MARTIN

TURNER

JAMES

Who TURNER works for:

TURNER

BLAKE

KING

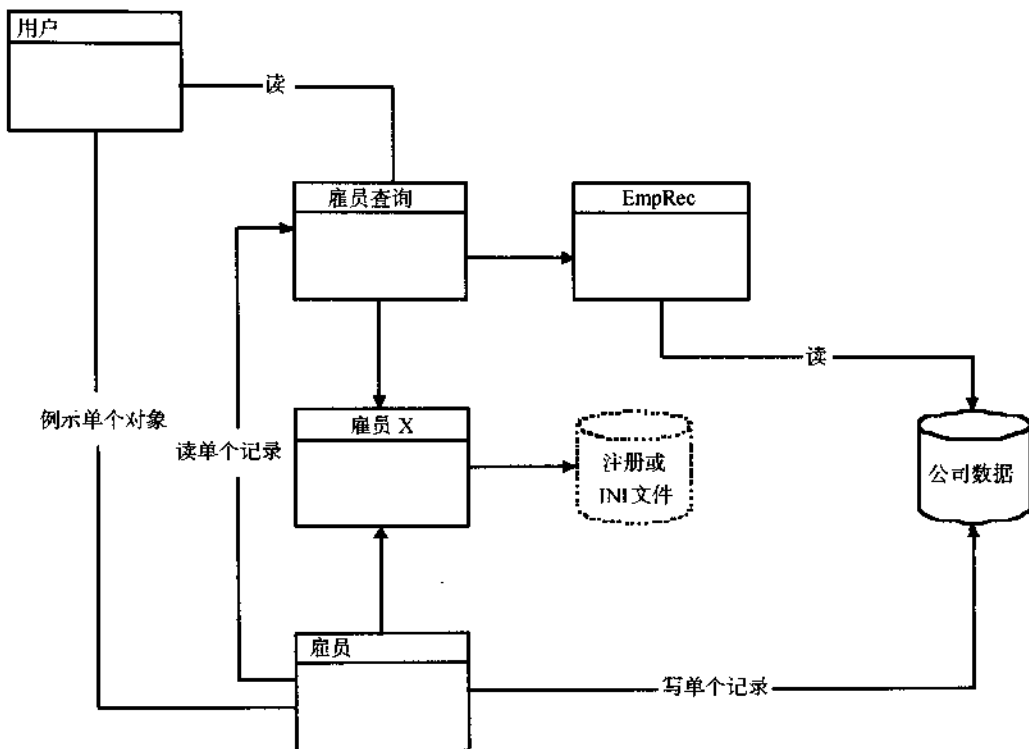


图 14.6 PXF 对象的相互作用

图 14.6 总结了基于 PXF 的用户对象是如何通过相互作用来提供永久性和查询能力的。访问 <http://www.freshsources.com/> 可获得完整的源代码。

14.10 小结

- 面向对象语言支持数据抽象、继承和多态。
- 在你的代码中继承使得一般和特殊关系明确。通过把共有代码置于基类中，继承可以减少工作量。继承还允许添加新的派生类而不影响已存在的代码。
- 公有继承表示基类和派生类之间的“is-a”关系。
- 指向基类的指针还可以指向派生类对象，派生类对象总是可以代替基类对象。
- 多态是指有关联的实现共用一个单独的接口，实现可以根据需要随着所包含的对象类型的变化而自动变化。C++通过虚函数支持多态。
- 虚函数通过函数指针表来模拟动态绑定。

第三部分

使用标准库

算 法

刚刚步入计算机程序设计领域的人，没有谁会否认算法是“计算机科学的要素”¹。程序用其机器周期对数据“做事情”。许许多多这样的“事情”被一次又一次地处理，因此我们给这些操作的集合取了一个名字，称之为算法。Webster 词典把算法 (algorithm) 定义为：

algorithm: 用于解决频繁涉及循环操作的数学问题 (比如找最大公约数) 的一种程序规则。

Knuth 在他的经典论述 *Fundamental Algorithms* 中，指出了欧几里德 (Euclid) 关于求两个整数的最大公约数的算法，它是众所周知的最古老的算法之一，定义为如下所示：

算法 E (Euclid 的算法)。已知两个正整数 m 和 n ，求它们的最大公约数，也就是说，求一个最大的能同时整除 m 和 n 的正整数。

E1. [找余数。] 用 n 除 m ，余数为 r 。

E2. [余数是 0 吗?] 如果 $r=0$ ，算法结束， n 就是答案。

E3. [相互交换。] 把 n 赋给 m ，把 r 赋给 n ，然后回到步骤 E1。

在 C 中这样的函数是：

```
/* 欧几里德 (Euclids) 算法 */
#include <assert.h>

int gcd(int m, int n)
{
    int r;
    assert(m > 0 && n > 0);

    while ((r = m % n) != 0)
    {
        m = n;
        n = r;
    }
}
```

```
    return n;
}
```

这个简单的程序足以说明算法的下面各个方面：

1. 有限性。换句话说，一个算法必须能够终止。中断计算机进程的能力是为我们有时不能确保这个重要特征的事实的证明！已知 m 和 n 都是正数， r 在 $[0, n]$ 范围内取值（即 $0 \leq r < n$ ）。由于 r 的当前值在下次循环中充当 n 的角色， r 的连续值形成了严格的递减序列，因此 r 的值最终必须达到 0，这确保了算法将会停止。

2. 输入。一个算法接受 0 个或多个的参数作为初始数据。参数通常伴随着限制条件或前提条件，为了使算法完成正确的功能必须满足这些条件。你可以像我在上面所做的那样使用断言（assertion）来验证这些条件，或者通过返回代码来设置一个出错条件，或者抛出一个异常来验证这些条件（关于这些技术的讨论，请见第 13 章）。

3. 输出。已知前提条件得到满足，一个算法约定以返回值传递某种数据、或者作为一个副作用来引发某种条件的产生，或两者兼而有之。副作用通常被称为后继条件。前提条件、后继条件以及返回的数据构成了一个算法与用户的协定。

15.1 复杂度

我们通常对一个算法的效率也很感兴趣。例如，虽然在一个数组内从头到尾地查找一个给定值总能得到想要的结果。但是，如果这是个已排序的序列，这样做就会很浪费时间（应该用二分法查找）。比较算法的一种方式就是衡量它们的复杂度，复杂度通常是指关键值操作的数量。考虑如下的线性查找函数：

```
// 线性查找：
int search(int* data, int n, int x)
{
    for (int i = 0; i < n; ++i)
        if (data[i] == x)
            return i;
    return -1;
}
```

这个函数的关键操作是把每一个元素与要查找的关键值 x 相比较。如果 $n > 0$ ，则查找循环将执行 1 到 n 之间的某个次数。如果元素不是按某一特定的顺序排列，那么对于任意给定的关键值，平均需要 $n/2$ 次比较，在最坏的情况下会有 n 次比较。我们把这称为“ n 阶”算法，并且我们称它的复杂度是“线性”的，因为操作的次数总是与 n 成比例。

算法复杂度的这一分类方法基本上是数学中“大 O”（big-oh）思想的应用。给定一个函数 $f(n)$ ，一个序列 x_n 是 $O(f(n))$ （读作“ n 的 f 的大 O”），若对于较大的 n 有：

$$|x_n| \leq m |f(n)|$$

其中 m 是常数。上面的线性查找函数是 $O(n)$ ，因为比较的次数总是由 n 限定（即 $f(n) = n, m=1$ ）。

现在考虑一个二分法查找算法:

// 二分法查找 (假设数据已按升序排序)

```
int search(int* data, int n, int x)
{
    int begin = 0;
    int end = n;
    while (begin < end)
    {
        int mid = (begin + end) / 2; // mid < end
        assert(mid < end);
        if (x == data[mid])
            return mid;
        else if (x < data[mid])
            end = mid - 1;
        else // x > data[mid]
            begin = mid + 1;
    }
    return -1;
}
```

这个 while 循环迭代了多少次? 每一此循环都将剩余的待查找数据的数量减少一半, 因此执行结果是 1 (即直到只剩下一个元素待检查) 时循环所进行的次数不超过 $n/2$ (使用整数运算)。小于或等于 n 时, 2 的最大幂应遵循下面的规则:

$$2^p < n$$

推出 $p \leq \log_2 n$ 。

因此, 迭代次数的最小上限将为 p 或 $p+1$, 这取决于 n 是不是 2 的幂。使用数学中上限的概念 (一个大于或等于已知数的最小整数), 迭代次数 i 遵循:

$$i < \lceil \log_2 n \rceil \quad // \log_2 n \text{ 的上限符号}$$

这样, 二分法查找就是 $O(\log_2 n)$ 。如果比较 $f(x) = x$ 和 $f(x) = \log_2 x$ 的函数曲线图形, 当 x 值较大时, 后者比前者小得多, 因此一个 $O(\log_2 n)$ 算法比一个线性算法更好。

我们期望最好的复杂性就是恒定次数 (即 $O(1)$)。例如, 在一个已排序的序列中查找最小值就是一个恒定次数算法, 这是因为不管表的大小如何它总是第 0 个元素。某种散列算法也展示了恒定次数的作用。(关于复杂性和散列的更多信息, 请参阅一本关于算法分析的好书³。)

按升序, 在通用的算法中最常见的衡量复杂度的方法是:

复杂度	代表性算法
$O(1)$	散列法、特殊算法
$O(\log n)$	二分法查找、其他树查找
$O(n)$	简单顺序遍历, 如线性查找
$O(n \log n)$	快速排序、堆排序
$O(n^2)$	简单排序算法 (如冒泡、插入)、各种矩阵算法和 <code>std::search</code>

为确保足够的性能，C++ 标准对标准库中算法的复杂度限制进行了规范。（见附录 A 中的算法列表）。

15.2 通用算法

标准 C++ 库包含了一个名为 `find` 的线性查找算法，可以采用如下方式使用它：

```
// find1.cpp: 查找一个整数数组
#include <algorithm>
#include <iostream>
using namespace std;
main()
{
    int a[] = {45,23,89,12,78};
    int n = sizeof a / sizeof a[0];
    int* past = a + n;

    int* p = find(a, past, 12);
    if (p != past)
        cout << "found " << *p << " in position "
              << p - a << endl;
    else
        cout << "item not found\n";
}
//输出:
found 12 in position 3
```

就像 `find` 一样，C++ 标准库中的大多数算法都在一定的范围内进行操作，比如上面的由指针 `a` 和 `past` 所限定的范围。第二个指针总是指向正在被处理序列结尾的下一个位置，这是从 ANSI C 借用过来的一个习惯用法，这样只要不废弃这个指针，它就总是指向序列结尾的下一个位置。算法 `find` 是一个通用算法，因为它能够处理几乎任何类型的序列。例如，可以采用如下方式查找字符串数组：

```
// find2.cpp: 查找一个字符串数组
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

main()
{
    string a[] = {"Albert", "Charles", "Horatio"};
    int n = sizeof a / sizeof a[0];
    string* past = a + n;

    string* p = find(a, past, "Charles");
```



```

    if (p != past)
        cout << "found " << *p << " in position "
            << p - a << endl;
    else
        cout << "item not found\n";
}

```

//输出:

```
found Charles in position 1
```

正如程序清单 15.1 中的程序所说明的那样,甚至可以在用户定义类型的数组中使用 `find`,只要这种类型支持一定的操作。`find` 为何会具有智能化的功能?因为它是一个函数模板。根据迄今为止我们所看到的,可以采用如下方式实现 `find`:

```

// find() 函数的一个可能的实现
template<class T>
T* find(T* start, T* past, const T& v)
{
    while (start != past)
    {
        if (*start == v)
            break;
        ++start;
    }
    return start;
}

```

编译器看到程序清单 15.1 中的表达式 `find(a,past,v)` 时,将实例化函数 `find(Person*,Person*,const Person&)`。从这一实现过程中可以看到, `find` 可以用于任何允许比较两个对象是否相等的类型(这就是我在程序清单 15.1 中包含 `operator==(const Person&,const Person&)` 的原因)。第 16 章将讨论标准 C++ 如何对它算法中的迭代机制进行分类,以及在标准库中它们是如何与各种容器一起使用的。本章在示例中只使用了数组容器。

有时候查找对象相等并不是确切想要的结果。例如,也许想找到比某个特殊值小的第一个元素,或当进行比较时,也许想让 `find` 仅仅考虑一个对象的子集,如单个的数据成员。程序清单 15.2 中的程序使用了 `find` 的一种版本,叫作 `find_if`,它使你可以提供一个确定查找成功的函数。函数 `isCharles` 是一个断言的例子,它只是一个返回值为布尔型的函数(或者函数对象,见下面)。函数 `isCharles` 是个一元函数,如果它的 `Person` 参数的 `name` 成员是字符串 "Charles" 则返回 `true`。算法 `find_if` 在它的范围内,依次将 `isCharles` 遍历应用于每一个对象直到该算法变为 `true` 为止。许多标准的算法都有这样的 `_if` 形式,为定制的匹配原则提供断言。

程序清单 15.1 查找用户定义类型数组

```

// find3.cpp
#include <algorithm>
#include <iostream>

```

```
#include <string>
using namespace std;

struct Person
{
    string name;
    int year;
    int month;
    int day;
    Person() : name("")
    {
        year = month = day = 0;
    }
    Person(const string& nm, int y, int m, int d)
        : name(nm)
    {
        year = y;
        month = m;
        day = d;
    }
};

bool operator==(const Person& p1, const Person& p2)
{
    return p1.name == p2.name && p1.year == p2.year &&
        p1.month == p2.month && p1.day == p2.day;
}

ostream& operator<<(ostream& os, const Person& p)
{
    os << '{' << p.name << ',' << p.month
        << '/' << p.day << '/' << p.year << ' ';
    return os;
}

main()
{
    Person a[3];
    a[0] = Person("Albert", 1901,1,20);
    a[1] = Person("Charles", 1897,3,11);
    a[2] = Person("Horatio", 1835,12,6);
    int n = sizeof a / sizeof a[0];
    Person* past = a + n;
    Person v("Charles", 1897,3,11);

    Person* p = find(a, past, v);
}
```

```

    if (p != past)
        cout << "found " << *p << " in position "
            << p - a << endl;
    else
        cout << "item not found\n";
}

//输出:
found {Charles,3/11/1897} in position 1

```

程序清单 15.2 使用断言查找一个序列

```

// find4.cpp: 使用断言
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

struct Person
{
    // 与清单 15.1 相同
};

bool isCharles(const Person& p)
{
    return p.name == "Charles";
}

ostream& operator<<(ostream& os, const Person& p)
{
    os << '{' << p.name << ',' << p.month
        << '/' << p.day << '/' << p.year << ' ';
    return os;
}

main()
{
    Person a[3];
    a[0] = Person("Albert", 1901,1,20);
    a[1] = Person("Charles", 1897,3,11);
    a[2] = Person("Horatio", 1835,12,6);
    int n = sizeof a / sizeof a[0];
    Person* past = a + n;

    Person* p = find_if(a, past, isCharles);
    if (p != past)

```

```

        cout << "found " << *p << " in position "
              << p - a << endl;
    else
        cout << "item not found\n";
}

```

15.3 函数对象

当然, `isCharles` 断言不是十分有用, 这是因为它只知道如何查找一个具体的关键值。除非能够提出一个更灵活的技术, 否则将不得不写出 `isAlbert`, `isHoratio`, 等等。你可能想将一个字符串参数加到 `isCharles` 中 (当然, 同时将它重新命名为类似 `byName` 的名字), 这样每次就可以自定义对 `find_if` 的调用, 但是 `find_if` 希望它的查找断言是一个带有具体参数类型的一元函数 (在此是 `Person`)。如果能够在空闲时构建函数, 那将是多么美妙呀, 但是, C++ 并不是 Lisp。解决的办法是应用通用的抽象规则: “当需要实现一个新的概念时, 创建一个新类。”⁴⁴ 你需要的是一个可以用查找关键值初始化并且起着一元函数作用的对象。“起函数作用”这个要求可以通过重载函数调用运算符而很容易得到满足。如果类 `X` 定义了 `operator`, 同时如果 `x` 是类 `X` 的一个对象, 那么表达式 `x` 就调用 `x.operator`。在 C++ 中, 这样类的实例称为函数对象。下面是一个函数对象的定义, 它将一个 `Person` 对象的 `name` 成员与一个给定的字符串进行比较:

```

struct byname
{
    string name;
    byName(const string& s) : name(s) {}
    bool operator()(const Person& p) {return p.name == name;}
};

```

现在对 `find_if` 的调用必须做相应的改变:

```
Person*p=find_if(a,past,byname("Charles"));
```

表达式 `byname("Charles")` 用它被初始化为 "Charles" 的 `name` 成员构造了一个临时的 `byname` 对象。`find_if` 的代码可能为如下形式:

```

template<class T, class Pred>
T* find_if(T* start, T* past, Pred p)
{
    while (start != past && !p(*start))
        ++start;
    return start;
}

```

其中 `Pred` 代表任何成员中带有 `bool operator (const T&)` 标记的类。表达式 `!p(*start)` 因而变为 `!byname("Charles").operator()(*start)`。

标准 C++ 库预定义了大量有用的函数对象 (见附录 C)。为了加以说明, 首先考虑

accumulate 算法，它是一种求一个序列元素之和的算法：

```
#include <numeric>    //定义 accumulate
```

```
...
```

```
int sum=accumulate(a,past,0);
```

有一个更通用的 **accumulate** 版本，它带有第四个参数，该参数表示一个任意的二元函数。如果要求得一个序列元素的乘积，而不是求和，就需要一个将其参数相乘以让 **accumulate** 使用的二元函数。用户可能总是自己编写这些代码，但是也可以考虑让库用 **multiplies** 函数对象来创建这个函数，如下所示：

```
#include <numeric>
```

```
#include <functional>    //定义函数对象
```

```
...
```

```
int prod=accumulate(a,past,1, multiplies<long>());
```

注意 **long** 模板参数的使用，因为可以想象得到乘积可能会溢出一个整型。除了下面代码中特殊的基类，对 **multiplies** 的定义是用户所期望的。

```
template<class T>
```

```
struct multiplies : binary_function<T, T, T>    // 下面解释
```

```
{
```

```
    T operator()(const T& x, const T& y) const
```

```
    {
```

```
        return (x * y);
```

```
    }
```

```
};
```

通过把函数对象适配器和函数对象一起使用，允许你定制和合并函数对象（见下面），就可以很好地完善算法断言使之达到一个详细程度很高的水平，而不需要自己编写函数。

15.4 函数种类

由于创建函数对象是经常要做的事情，所以标准库有一些特性，这些特性使得做这个工作更加容易。但是首先理解下面 C++ 对函数的分类是很有帮助的：

1. 发生器——没有参数并返回单值的函数。
2. 一元函数——带有一个任何类型的参数并返回单值的函数。
3. 多元函数——带有两个任何（可能不同）类型的参数并返回单值的函数。

标准随机数函数 **rand** 是一个很好的发生器的例子。然而，在大多数时候，用户更关心一元函数和多元函数。为了利用标准 C++ 库中所有的函数对象机制，需要使用可改写的一元和二元函数对象，它们使用了嵌套的 **typedef** 来描述参数和返回类型。**byName** 函数对象的一个可改写的版本如下：

```
struct byname
```

```
{
```

```
    typedef bool return_type;
```

```

typedef Person argument_type;
string name;
byName(const string& s) : name(s) {}
bool operator()(const Person& p) {return p.name == name;}
};

```

然而有一个名为 `unary_function` 的类模板，它为用户记录了这些类型，因此可以从它来派生，没有必要再键入 `typedef` 语句：

```

template<class ArgType, class RetType>
struct unary_function
{
    typedef ArgType argument_type;
    typedef RetType result_type;
};

```

正如在前面 `multiplies` 的定义中看到的那样，也存在一个 `binary_function` 模板，该模板记录了这些类型：`first_argument_type`、`second_argument_type` 和 `result_type`。现在一个可改写的 `byName` 可以定义如下：

```

struct byName : unary_function<Person, bool>
{
    string name;
    byName(const string& s) : name(s) {}
    bool operator()(const Person& p) {return p.name == name;}
};
// argument_type == Person, result_type == bool

```

利用可改写的函数对象，用户可以在需要的时候使用函数对象适配器来创建复杂的函数对象。

15.5 函数对象适配器

一个函数对象适配器是一个函数模板，它把一个或更多的函数对象以及其他可选的值作为参数并且返回一个新的函数对象（关于其完整的列表，见附录 C）。例如适配器 `not1` 逻辑地将它的一元判断参数的判断取反，因此下面的语句将以一个不同于“Charles”的名字在数组中查找第一个 `Person` 对象：

```
Person* p=find_if(a,past,not(byname("Charles")));
```

从根本上说，适配器 `not1` 是一个包装封面，它调用其参数并在返回之前将其结果取反。

通过把一个已知值保存为第二个参数，适配器 `bind2nd` 将一个二元函数对象转换为一元函数对象。例如为了查找一个特殊的 `Person` 对象，可以结合 `bind2nd` 和 `equal_to` 来形成一个合适的判断：

```
Person v("Charless",1897,3,11);
Find_if(a,past,bind2nd(equal_to<Person>(),v));
```

可修改的二元函数对象 `equal_to<Person>` 由预定义模板 `equal_to` 实例化，而且它将两个 `Person` 对象作为参数。`bind2nd` 适配器创建了一个存储 `v` 值的一元函数对象。无论什么时候，

当 `find_if` 调用以一个来自数组的 `Person` 对象作为参数的新函数对象的 `operator` (`const Person&`) 时, 该函数对象就依次调用以 `v` 作为第 2 个参数的 `equal_to<Person>`。这样 `a` 中每一个 `Person` 对象都和 `v` 进行比较, 直到找到一个相匹配的为止。还有一个 `bind1st` 适配器, 在这种情况下, 它可以同样工作, 这是因为比较相等是可以互换的。

上面的例子需要一个完整的 `Person` 对象作为一个查找关键值。为了把查找简化成为仅仅是同前面一样的名称字符串的比较, 可以使 `byName` 成为一个二元断言并且使用 `bind2nd`, 如下面的语句所示:

```
struct byname:binary<Person,string,bool>
{
    bool operator()(const Person& p, const String& s ) const
    {
        return p.name == s;
    }
};
```

注意这个 `byName` 的版本不必存储任何数据 (因为 `bind2nd` 已经存储数据了)。

如果这还不够简单, 那么还有另一个适配器 `ptr_fun`, 它由一个函数指针构建一个函数对象, 因此所有真正要做的也就是提供一个开始的比较函数, 如:

```
bool byname(const Person&p,const string&s)
{
    return p.name===s;
}
```

然后像下面这样调用 `find_if`:

```
find_if(a,past,bind2nd(ptr_fun(byName), "Charles"));
```

适配器 `ptr_fun` 推断出它的参数是一元还是多元函数, 并且返回一个合适的可修改的函数对象。

15.6 算法种类

正如在附录 C 中所看到的, C++ 库标准中有大量的通用算法, 每一种通用算法都属于下面三组算法中的一组:

1. 通用序列算法, 它对对象序列执行常见的操作 (例如 `find`、`find_if`、`count`、`copy`、`transform` 等)。
2. 有关排序的算法, 比如 `sort` 和 `partial_sort`, 重新将一个序列中的元素排序, 还有其他算法像 `min` 和 `max`, 返回关于次序的信息。
3. 数值算法, 它完成数学操作, 比如 `accumulate` 和 `inner_product`。

头文件 `<algorithm>` 声明了前两组中算法的函数模板, 而第三组算法的函数模板在头文件 `<numeric>` 中声明。程序清单 15.3 中的程序举例说明了所选的几种算法 (输出在程序清单 15.4 中)。

程序清单 15.3 举例说明各种算法

```
// algs.cpp: 举例说明不同的算法
#include <algorithm>
#include <numeric>
#include <iostream>
#include <string>
#include <utility>
using namespace std;

// 打印任何数列的函数模板:
template<class T>
void print_array(T* begin, T* end)
{
    while (begin != end)
        cout << *begin++ << ' ';
    cout << endl;
}

// 测试整型数据奇偶性的函数:
inline bool odd(int n)
{
    return n % 2 == 1;
}

string parity[] = {"even", "odd"};

inline void print_parity(int n)
{
    cout << parity[n%2] << ' ';
}

// 如果是奇数则返回 1; 偶数则为 0:
inline int calc_parity(int n)
{
    return n%2;
}

main()
{
    int a[] = {1,2,3,4,5};
    const int nelems = sizeof a / sizeof a[0];
    int b[nelems];

    // 将 a 拷贝到 b (必须有足够的空间)
    copy(a, a + nelems, b);
    cout << "b == ";
```



```

print_array(b, b+nelems);

// 进行等同性检测:
cout.setf(ios::boolalpha);
bool test = equal(a, a + nelems, b);
cout << "a == b? " << test << endl;

// 对 b 进行逆转:
cout << "Reverse b: ";
reverse(b, b + nelems);
print_array(b, b+nelems);
test = equal(a, a + nelems, b);
cout << "a == b? " << test << endl;

//对 b 进行排序:
sort(b, b + nelems);
print_array(b, b+nelems);
test = equal(a, a + nelems, b);
cout << "a == b? " << test << endl;

//对 a 的元素求和:
int sum = 0;
cout << "sum == "
    << accumulate(a, a + nelems, sum)
    << endl;

//在 b 中以 3 计数:
int n3 = count(b, b + nelems, 3);
cout << "# = 3's in b: " << n3 << endl;

//计算 b 中奇数:
int nodd = count_if(b, b + nelems, odd);
cout << "# odd's in b: " << nodd << endl;
cout << endl;

// 用奇偶值取代 b:
for_each(a, a + nelems, print_parity);
cout << endl;
transform(b, b + nelems, b, calc_parity);
print_array(b, b+nelems);
}

```

程序清单 15.4 程序清单 15.3 的输出

```

b == 1 2 3 4 5
a == b? true

```

```
Reverse b: 5 4 3 2 1
a == b? false

1 2 3 4 5
a == b? true

sum of a == 15

# = 3's in b: 1
# odd's in b: 3

odd even odd even odd
1 0 1 0 1
```

15.7 小结

- 算法是计算机科学的“要素”。
- C++标准库中的算法是函数模板，它们中的大多数都是处理由[begin,end]所限定范围的一系列对象，其中 end 指向最后要被处理的元素的下一个位置。
- 标准算法分为三大类：通用序列操作、有关排序的操作和数值操作。
- 函数对象是一个定义运算符的类的实例。
- 可改写的函数对象含有对它的参数和返回值类型的定义。
- 标准函数对象实现发生器、一元函数和二元函数。
- 判断是一个返回布尔类型的函数（或函数对象）。
- 函数对象适配器是一个函数模板，它把一个或多个函数对象以及其他可选的数值作为参数，并且返回一个新的函数对象。
- 通过把函数对象和函数对象适配器结合，可以自定义算法断言，通常没有必要自己编写函数。

15.8 参考文献

1. Sedgewick, Algorithms, Addison-Wesley, 1983, p. 4.
2. Knuth, *The Art of Computer Programming*, Second Edition, Vol. 1, TFundamental Algorithms, Y Addison-Wesley, 1973, p. 2.
3. 参阅, Aho et al., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
4. 参阅 Koenig, *Ruminations on C++*, Addison-Wesley, 1997, p. 151. 中文版《C++沉思录》人民邮电出版社, 2002, 10。

容器和迭代器

容器有时也称为集合，是一个容纳其他对象的对象。因此，从有限的意义上讲，结构就是容器，但是大多数容器的有用类型都允许在运行期增加成员，并且允许以某种顺序遍历对象的集合。这些年来，应用最为广泛的容器当然是数组。当需要由对象的位置来直接访问一个对象时，使用数组就特别方便。然而，当需要在不是末尾的位置上添加或删除元素时，它们又不是那么方便。当然，数组的另一个缺点就是必须事先指明数组最多所能容纳元素的个数。为了克服容量的问题，标准库定义了一个向量模板（vector），它实例化了一个类似于数组的容器，你可以根据需要在其中想增加多少元素就增加多少元素。

下面的程序用字符串向量在文本文件中单独地保存标记：

```
// vec1.cpp

#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
using namespace std;

main()
{
    vector<string> tokens;
    string token;

    //读符号；填向量
    while (cin >> token)
        tokens.push_back(token);
    int ntok = tokens.size();
    cout << "There are " << ntok << " tokens:\n";
}
```

```
    for (int i = 0; i < ntok; ++i)
        cout << i << ": " << tokens[i] << endl;
}
// 输入:
how now brown cow
```

```
// 输出:
There are 4 tokens:
0: how
1: now
2: brown
3: cow
```

虽然这一行:

```
vector<string>tokens;
```

把 `token` 声明为一个字符串向量，但是可以声明任何具体类型的向量。成员函数 `push_back` 把一个新元素附加到这个向量的末尾。

为了进一步说明向量模板的使用，考虑标准 C/C++ 的命令行参数工具。通常获得的参数是用户通过命令行输入给主函数 `main` 的参数:

```
main(int argc, char*argv[])
```

如果你的用户用长参数列表重复地调用程序，他们也许更愿意把参数标记存储在一个文件中，因此当他们输入命令行:

```
prog @arg_file
```

“at”符号 (@) 告诉你在文件 `arg_file` 中查找这些标记。总之，既然你正在打开文件，为什么不让它们成为嵌套文件呢？例如，如果 `arg_file` 包含了:

```
arg1
@file2
arg2
```

并且 `file2` 包含:

```
arg3
arg4
```

那么，在命令行中输入 `prog @arg_file` 就应该和下面的用户命令等价。

```
prog arg1 arg3 arg4 arg2
```

使用向量 `vector`，可以很容易地创建一个类，我们叫它 `Arglist` 类，它自动扩展下列的间接文件:

```
#include <string>
#include <vector>
using std::string;
using std::vector;
```

```
class Arglist
{
public:
```

```

    Arglist(size_t, char **);
    size_t count() const;
    const string& operator[](size_t) const;
private:
    vector<string> args;    // 保存 args
    void expand(char *);    // 读文件
    void add(char *);       // 添加一个 arg
};

```

Arglist 对象用一个字符串向量，这个字符串向量通过处理由它们的参数表示的 `argc/argv` 来完成初始化，并在需要时打开文件。下面的程序用 **arglist** 类来打印它的扩展参数列表，每行一个参数：

```

#include <iostream>
#include "arglist.h"

main(int argc, char* argv[])
{
    Arglist args(--argc, ++argv);    // 忽略程序名
    for (int i = 0; i < args.count(); ++i)
        cout << "arg[" << i << "] == " << args[i] << endl;
}

```

Arglist 类五个成员函数中的三个把它们的责任委托给向量 **args**：

```

size_t Arglist::count() const
{
    return args.size();
}

const string& Arglist::operator[](size_t i) const
{
    return args[i];
}

void Arglist::add(char* arg)
{
    args.push_back(arg);
}

```

有趣的工作在 **expand** 中完成，它处理文件中的每一个参数，并在需要的时候递归地打开嵌套文件：

```

#include <fstream>
using namespace std;

void Arglist::expand(char* fname)
{
    ifstream f(fname);
    char token[64];

```

```

while (f >> token)
    if (token[0] == '@')
        expand(token+1);
    else
        add(token);
}

```

Arglist 的构造函数只调用 expand 或者 add, 这取决于用户输入的原始参数是否有一个前缀 @ 字符:

```

Arglist::Arglist(size_t arg_count, char **arg_vec)
{
    for (int i = 0; i < arg_count; ++i)
        if (arg_vec[i][0] == '@')
            expand(arg_vec[i]+1);
        else
            add(arg_vec[i]);
}

```

16.1 标准容器

标准库提供了三类容器: 基本序列容器、容器适配器和关联容器。基本序列、向量、表和队列是具有自身独特特性的有顺序的数据结构, 向量和双端队列 (deque) (是 “double-edged queue” 的缩写, 读做 “deck”) 是为随机访问而优化了的类似于数组的数据结构, (即它们为任意序列元素提供时不变的访问, 并重载了运算符 []). 向量只在序列的结尾提供了时不变的元素的插入和删除, 而双端队列可以在序列的开头和结尾完成上述工作。表是一个双向链表数据结构, 它在序列的任意处都支持时不变的插入和删除, 当插入和删除操作不是发生在队列的结尾时, 特别是当需要连续地访问时, 表是首选的容器。

所有标准容器类库都有值语义, 这意味着它们都有运算符和构造函数, 这些运算符和构造函数提供比较 (至少比较是否相等)、按值传递并返回和赋值等操作。更重要的是它们希望自己包含的对象具有值的语义, 因为它们保存所给的对象拷贝并且在需要的时候例行公事地对这些对象进行拷贝和赋值。所有的容器也都定义了如下的成员函数:

```

size_type size() const;           // 元素的当前数
size_type max_size() const;       // 可能元素的最大数
bool empty() const;               // size() 是否等于 0

```

size_type 是无符号整型, 它的操作与 size_t 相似, 并且由每个容器适当地定义。顺序容器也定义了在任何位置插入和删除元素的函数, 以及下面的操作 (T 是被包含的对象的类型):

```

void resize(size_type, T=T());    // 扩展或者截断
T& front() const;                 // 返回第一个元素
T& back() const;                  // 返回最后一个元素
void push_front(const T&);        // 在开头插入 (向量不支持这个操作)
void push_back(const T&);         // 在结尾附加
void pop_front();                 // 删除第一个元素 (向量不支持这个操作)

```

```
void pop_front();           //删除最后一个元素
T& at(size_type n);        //返回第 n 个元素（表不支持这个操作）
```

你或许感到疑惑，为什么表不提供成员函数来操纵它的序列呢？这个问题的答案也解释了为什么标准容器显式地定义了少数几种标准算法，如第 15 章所讨论过的算法。答案就是：迭代器。

16.2 迭代器

迭代器是支持序列遍历的一种指针的概括，例如，要访问列表中的每个元素，可以像下面这样做：

```
list<T> lst;
// 插入某些元素，然后：
list<T>::iterator p = lst.first();
while (p != lst.end())
{
    // 处理当前元素(*p)，然后：
    ++p;
}
```

所有容器定义了类型迭代器和常迭代器（`const_iterator`），它们支持拷贝、赋值、比较是否相等操作以及至少对运算符*、++（两个特性* 和 ++）和->的重载，因而可以用一般的指针语法处理一个序列。成员函数 `first` 返回一个指向第一个元素的迭代器，而成员函数 `end` 指向最后一个元素的下一个位置，这样当需要时可通过标准算法定义一个范围。例如，为了计算整数表中 3 的所有实例的个数，可使用 `count` 函数，如下所示：

```
int n=count(lst.first(),lst.end(),3);
标准库中的算法是根据迭代器的术语而被实际指定的。例如 find 算法，实现如下：
template<class Iterator, class T>
Iterator find(Iterator begin, Iterator end, const T& v)
{
    while (begin != end)
    {
        if (*begin == v)
            break;
        ++begin;
    }
    return end;
}
```

正是将包容、迭代和算法安排成独立的概念，并结合模板的灵活性，才使得标准 C++ 库成为我们曾设想过的功能最强大的、通用的库。算法既适用于系统预定义数组序列也适用于复杂的容器。实质上，容器可以管理任何具体类型（甚至其他容器）的集合。迭代器是连接容器和算法的粘结剂。

注意 `find` 函数希望迭代器只支持三种运算符：`!=`、`*`和`++`。其他算法可能需要其他的特

征，如运算符`--`用来进行反向遍历，或运算符`[]`用来进行随机访问。标准库将迭代器分成五种不同的种类，因此一个算法能公布它的迭代要求，或可以提供算法的不同版本以适应不同的情况。

16.3 迭代器种类

五种迭代器分别是：

1. 输入迭代器；
2. 输出迭代器；
3. 前向迭代器；
4. 双向迭代器；
5. 随机访问迭代器。

输入和输出迭代器支持拷贝、比较是否相等（运算符`==`、`!=`）、间接引用（运算符`*`）以及由运算符`++`和`++(int)`进行的单向遍历。它们的区别在于让用户如何处理间接引用的对象：输入迭代器产生一个只读引用而输出迭代器产生一个只写引用。如同许多其他的标准算法，`find` 要求有一个输入迭代器。出于编写文档的考虑，按如下方法实现将更有效。

```
template<class InputIterator, class T>
InputIterator find(InputIterator begin, InputIterator end, const T& v)
{
    while (begin != end)
    {
        if (*begin == v)
            break;
        ++end;
    }
    return end;
}
```

前向迭代器允许能够进行读和写地多路遍历序列，因而它既能代替输入迭代器，也能代替输出迭代器。例如，算法 `unique` 从序列中删除了除第一个连续的副本元素外的所有元素。调用

```
list<T>::iterator p=unique(lst.first,lst.end());
```

通过把序列中不是副本的元素向上移动，并在遍历过程中覆盖副本，返回一个用来限定剩下对象序列的超过末尾的迭代器。由于需要读写 `lst` 中的元素，因而需要前向迭代器。如果提供一个不能胜任这一功能的迭代器，则将显示缺少操作的编译错误。

双向迭代器能代替前向迭代器，但是它通过运算符`--()`和`--(int)`也支持后序遍历序列的操作，因此，后序遍历一个表的过程如下：

```
// 用运算符--后序遍历：
list::iterator p<T> = lst.end();
while (p > lst.begin())
```



```

{
    --p;                      // “接近的” 后向
    // 处理*p, 然后:
    if (p == lst.begin())
        break;
}

```

支持双向迭代器的容器也定义了成员函数 `rbegin` 和 `rend`，它们返回的是反向迭代器，这为后向地遍历一个表中的元素提供了更好的方法，即

```

//由反向迭代器后序遍历:
list::reverse_iterator<T> p = lst.rbegin();
while (p != lst.rend())
{
    //处理*p, 然后:
    ++p;                      // “接近的” 后向
}

```

正如希望的那样，反向算法要求一个双向迭代器，因为最有效的普通实现使用两个迭代器，在序列的每端各一个，它们向相对的方向移动直到在序列中间相遇为止。

随机访问迭代器是一个双向迭代器，用常用的运算符：`+`、`++`、`-`、`--`、`[]`、`<`、`<=`、`>`和`>=`来模仿指针运算。通常的排序算法要求有随机访问迭代器，因为需要交换元素以重新排列它们。由于表不能有随机访问迭代器，因此它有其自己的排序成员函数。总之，如果一个通用算法既不能应用于一个容器又不能优化容器的效率，则它就以定制的版本作为容器的成员函数出现。其他关于表的优化算法包括 `merge`、`remove` 和 `unique`。

16.4 特殊用途的迭代器

在第 15 章中，所有的例子都采用覆盖方式向数组序列中写元素，这意味着目标数组必须有足够的空间来接受输出。下面的例子，试图把 `vector<int>` 中的元素拷贝到一个空的整型向量中去，它将在运行期时失败：

```

vector<int> v1;
// 填充 v1, 然后:
vector<int> v2;

copy(v1.begin(), v1.end(), v2.begin());    // 错误

```

幸运的是，有一种办法可以让标准容器在插入模式下进行操作以允许在已有的序列中的任何位置插入一个新对象的算法。插入迭代器是一个输出迭代器，它为其容器用一个适当的插入操作来替换所有对运算符`*`的调用。函数模板 `back_inserter` 为自己的容器参数创建这样的迭代器，从而用对 `push_back` 的调用来替换所有的输出操作。下面的 `copy` 语句修正了前面例子的问题：

```

copy(v1.begin(), v1.end(), back_inserter(v2));

```

还有一个 `front_inserter` 模板，当然它只对双端队列和表起作用，因为向量不支持 `push_front`。一个更通用的版本——`inserter`，在一个给定的位置上按照先入先出的顺序插入一个对象，例如：

```
//在 lst2 的第 n 个元素之前开始插入对象：
```

```
copy(lst1.begin(), lst1.end(), inserter(lst2, lst2.begin()+n));
```

除此之外还有为 I/O 流而设定的特殊迭代器。`Ostream_iterator` 是一个输出迭代器，它能向输出流插入对象。例如，下面的语句把 `vector<int>` 中的元素打印到 `cout` 中，每个元素之间用空格分开：

```
copy(v1.begin(), v1.end(), ostream_iterator<int>(cout, " "));
```

`istream_iterator` 是一个输入迭代器，它从输出流中读元素。例如，为了繁殖一个整型向量，可以用 `copy` 函数，如下所示：

```
copy(istream_iterator<int>(cin), istream_iterator<int>, back_inserter(v1));
```

由一个空的参数表创建的 `istream_iterator`，当到达文件结尾时作为一个超过结尾的迭代器使用。

程序清单 16.1 中的程序说明了上面所讨论的概念。它用向量的向量，特别是用 `vector<vector<char>>` 来存储表示 x - y 平面中第一象限的字符的坐标格，在坐标格中每个 `vector<char>` 是一行，它对应于一个固定的 y 值。行的定位使得水平打印变得很方便，但是它需要一个违反直觉的 y - x 索引机制（相对于 x - y ），因此语句：

```
grid[f(x)][x] = '*'; //y 坐标是第一个
```

程序清单 16.1 用向量的向量来产生一个图表

```
// grid.cpp: 在网格字符中存储 x-y 图表
```

```
#include <vector>
```

```
#include <iostream>
```

```
#include <iterator>
```

```
using namespace std;
```

```
namespace
```

```
{
```

```
    typedef vector<char> row_type;
```

```
    typedef vector<row_type> grid_type;
```

```
    const int xmax = 15;           //列数
```

```
    const int ymax = xmax;         //行数
```

```
}
```

```
main()
```

```
{
```

```
    void print_grid(const grid_type&);
```

```
    double f(double);              //图表函数
```

```
    grid_type grid;                //行的向量
```

```

// 初始化 y 轴并清除第一象限:
grid.reserve(ymax);
row_type blank_row(xmax);
blank_row[0] = '|';
for (int y = 0; y < ymax; ++y)
    grid.push_back(blank_row);

// 初始化 x 轴:
grid[0][0] = '+';
for (int x = 1; x < xmax; ++x)
    grid[0][x] = '-';

// 增加 f() 的点:
for (int x = 0; x < xmax; ++x)
    grid[f(x)][x] = '*';           //面向行

print_grid(grid);
}

double f(double x)
{
    // 强迫固定在网格内部!
    return x * x / ymax + 1.0;
}

void print_grid(const grid_type& grid)
{
    grid_type::const_reverse_iterator yp;
    for (yp = grid.rbegin(); yp != grid.rend(); ++yp)
    {
        //打印一行:
        copy(yp->begin(), yp->end(), ostream_iterator<char>(cout, " "));
        cout << endl;
    }
}

//输出:
|           *
|
|           *
|
|           *
|           *
|           *
|
|           *

```

```

|      *
|      *
|      *
|      *
|      *
|      **
|      ***
|      ****
+-----+

```

为避免范围上的错误，在这个例子中用来画图的函数上附加了条件以满足在坐标格所限定的范围之内。函数 `vector::reserve` 要求立刻为给定数量的元素分配足够的空间以避免在一次插入一个元素时所发生的由重新分配内存造成的资源浪费。

16.5 容器适配器

容器适配器、队列（queue）、栈（stack）以及优先权队列（priority_queue），都是用一个序列来实现的高级的数据结构。例如，一个栈可以基于双端队列、表或向量，这取决于如何声明它：

// 用下面的向量声明一个字符串栈：

```
stack<string, vector<string>> string_stack;
```

如果省略了第二个模板参数，`queue<T>` 和 `stack<T>` 使用 `deque<T>` 作为第二个模板参数，而 `priority_queue<T>` 用 `vector<T>` 作为第二个模板参数，也可以通过把一个已有的序列作为构造函数的参数来使用它：

```
deque<int> d;
```

// 繁殖 d，然后：

```
queue<int> q(d);           //把 d 看作一个队列
```

假如容器能提供适配器所需要的成员函数，则可以用自己设计的容器作为支持序列。

这些容器类被称为适配器是因为它们通过为用户提供更有限的接口从而使得一个序列适合特定的应用（特别地，它们不提供迭代器）。例如，堆栈只有后入先出结构所要求的成员函数：

```
bool empty() const;
```

```
size_type size() const;
```

```
value_type& top();
```

```
const value_type& too() const;
```

// value_type 包含在 type(T) 中

```
void push(const value_type& x);
```

```
void pop();
```

这些函数仅仅调用了基本序列的相应操作（例如 `pop` 调用 `pop_front` 操作）。虽然队列定义了 `front` 和 `back`，而不是 `tip`，而且它的 `push` 方法对基本序列进行了 `push_back`，但是 `pop` 还是如希望的那样调用 `pop_front`。由于 `vector` 不支持 `pop_front`，因此它也不能支持一个队列适配器。`Priority_queue` 实现了一个典型的堆数据结构，它的 `pop` 方法返回一个关于某些断言的最小值（默认时是 `less<T>`）。

16.6 关联容器

关联容器就是集合 (set)、多重集合 (multiset)、图 (map) 和多重图 (multimap)。集合是一个包括独特的关键词并支持插入和测试成员资格操作的容器。图也称为词典或关联数组，它通过把特殊的关键词和同伴对象进行配对并在一些数据结构中存储这些匹配对来使它们结合。关联容器、多重集合和多重图不要求关键词是唯一的，并提供方法来遍历相等关键词的多个值。

尽管顺序不是这些数据结构固有的性质，但是它们根据作为构造函数的参数 (缺省时 `less<T>`) 随意提供的断言来对它们的元素进行排序，因此可以把它们看成是序列。标准库也规定了可以用对数次来查找关联容器的关键词 (见第 15 章中的第一部分——“复杂性”)。一个典型的实现使用树结构的某种类型来满足这些要求。这些容器也提供基本算法的自定义版本，如 `find` 和 `count`，它们操作起来比通用版本更为有效。

程序清单 16.2 中的程序说明了一个字符串集。注意到试图两次插入 “Tennessee” 的操作都被忽略了。在程序清单 16.3 中的图例子说明了插入匹配对的两种方法。一种方法是调用一个带有匹配对的 `insert` 成员函数。匹配模板 (带有 `first` 和 `second` 成员) 是标准库的一部分，在这种情况下 `element_type` 是 `pair<string, string>`。另外一种方法是用 `[]` 运算符，它带有一个关键词作为索引。注意这两个例子中的顺序是在词典编纂中用的升序排列，因为默认的顺序判断是 `less<string>`。同时也要注意 `map<string>::operator[]` 用 “Knoxville” 改写了 “Nashville”，另一方面，一个已存在的关键词的插入调用，将被忽略。

16.7 应用

通过研究实际的例子，标准库的作用得到了最好的体现。程序清单 16.4 包含了一个文件指令程序。它读取文本行并在标准输出中回显，除了以标记 `#include` 开头的行之外。对于这些行，程序替换了由下一个标志所显示的文件内容，就像 C/C++ 预处理器所做的那样，并且为嵌套文件递归地重复该过程，但是只针对那些当前没有被处理的文件。该程序用一个集来跟踪活动的文件。

程序清单 16.5 中的程序是一个前后对照表发生器。它从文本文件中读取标记，并跟踪每一个标志出现的行数。由于标志仅需要被列出一次，它们的行数也一样，我用一个图来对字符串和一组行数进行配对。注意内部的 `while` 循环繁殖了图。行

```
m.insert(make_pair(token, val_type()));
```

在标记第一次出现的地方插入一个空集，随后的赋值被忽略了。下一行重新得到和标志关联的集，然后把它作为左值来储存新行数 (但是副本又一次被忽略了)。库函数模板 `make_pair` 在空闲中创建了匹配对。

程序清单 16.2 说明了一个字符串集

```
// set.cpp
#include <iostream>
#include <set>
#include <string>
using namespace std;

main()
{
    //增加一个集:
    set<string> s;
    s.insert("Alabama");
    s.insert("Georgia");
    s.insert("Tennessee");
    s.insert("Tennessee");

    //打印出来:
    set<string>::iterator p = s.begin();
    while (p != s.end())
        cout << *p++ << endl;    cout << endl;

    //做查找:
    string key = "Alabama";
    p = s.find(key);
    cout << (p != s.end() ? "found " : "didn't find ")
        << key << endl;

    key = "Michigan";
    p = s.find(key);
    cout << (p != s.end() ? "found " : "didn't find ")
        << key << endl;
}

//输出:
Alabama
Georgia
Tennessee
found Alabama
didn't find Michigan
```

程序清单 16.3 把字符串映射到字符串中

```
// map.cpp
#include <iostream>
#include <map>
#include <string>
```

```

using namespace std;

main()
{
    // 便利的 typedef:
    typedef map<string, string, greater<string>()> map_type;
    typedef map_type::value_type element_type;
    // 插入一些元素(两种方法):
    map_type m;
    m.insert(element_type(string("Alabama"), string("Montgomery")));
    m["Georgia"] = "Atlanta";
    m["Tennessee"] = "Nashville";
    m["Tennessee"] = "Knoxville";

    // 打印映像:
    map_type::iterator p = m.begin();
    while (p != m.end())
    {
        element_type elem = *p++;
        cout << '{' << elem.first << ',' << elem.second << "}\n";
    }
    cout << endl;

    // 由关键字而重新得到:
    cout << "'" << m["Georgia"] << "'" << endl;
    cout << "'" << m["Texas"] << "'" << endl;
}

// 输出:
{Alabama, Montgomery}
{Georgia, Atlanta}
{Tennessee, Knoxville}
"Atlanta"
""

```

程序清单 16.4 一个递归安全文件包含程序

```

// include.cpp: 嵌套文件包含
#include <iostream>
#include <fstream>
#include <sstream>
#include <set>
#include <string>
using namespace std;

// 活动文件名的列表:

```

```
set<string> files;
void include(const string&);

main(int argc, char* argv[])
{
    if (argc > 1)
        try
        {
            include(argv[1]);
        }
        catch (string& s)
        {
            cerr << s << endl;
        }
}

void include(const string& fname)
{
    // 打开文件; 增加列表:
    ifstream f(fname.c_str());
    if (!f)
        throw string("error opening file: ") + fname;
    files.insert(fname);

    // 过程文件:
    string line;
    while (getline(f, line, '\n'))
    {

        // 检测第一个符号:
        istream is(line);
        string word;
        is >> word;

        if (word == "#include")
        {
            // 试图包括嵌套的文件:
            string nested;
            is >> nested;
            if (files.find(nested) == files.end())
                include(nested);

            // 忽略剩下的文件
        }
        else
    }
```



```

        cout << line << endl;
    }

    //从列表中删除文件名:
    files.erase(fname);
}

```

程序清单 16.5 前后对照表发生器程序

```

// xref.cpp: 打印每个命令发生的所在行的行数
#include <iostream>
#include <iomanip>
#include <string>
#include <set>
#include <map>
using namespace std;

namespace
{
    typedef set<int> val_type;
    typedef map<string, val_type> map_type;
    const int WORD_WIDTH = 15;
}

main()
{
    string next_token(const string&, int&);
    void print_list(const val_type&);

    map_type m;
    string line;
    int lineno = 0;

    //读每一行:
    while (getline(cin, line, '\n'))
    {
        ++lineno;

        //处理行中每一个命令:
        int pos = 0;
        string token;
        while (!(token = next_token(line, pos)).empty())
        {
            //增加图:
            m.insert(make_pair(token, val_type()));
        }
    }
}

```

```

        m[token].insert(lineno);
    }
}

//打印结果:
cout << "No. of distinct words: " << m.size() << endl;
cout.setf(ios::right, ios::adjustfield);
for (map_type::const_iterator p = m.begin(); p != m.end(); ++p)
{
    cout << setw(WORD_WIDTH) << p->first << setw(0) << ": ";
    print_list(p->second);
    cout << endl;
}

void print_list(const val_type& v)
{
    const int NUM_WIDTH = 5;
    const int INDENT = WORD_WIDTH + 2;
    const int NUMS_PER_LINE = 8;

    val_type::const_iterator p = v.begin();
    for (int i = 0; p != v.end(); ++i, ++p)
    {
        cout << setw(NUM_WIDTH) << *p;

        //如果需要, 开始一个新的行:
        if ((i+1) % NUMS_PER_LINE == 0 && i < v.size()-1)
        {
            cout << endl << setw(INDENT) << ' ';
        }
    }
}

string next_token(const string& s, int& pos)
{
    static const string bad_chars =
        " \t\n\v\a\f`~!@#%&^*()=+[{("
        ")}\|\|;:'\"<.>/?-1234567890";

    // 解压缩下一个*bad_chars*限定的符号:
    int begin = s.find_first_not_of(bad_chars, pos);
    int end = s.find_first_of(bad_chars, begin);
    pos = end;
    if (begin == string::npos)

```

```

        return string("");
    else if (end == string::npos)
        return s.substr(begin);
    else
        return s.substr(begin, end-begin);
}

```

程序清单 16.6 中的程序是 UNIX 的 `unique` 过滤器的变种, 它从文本文件中删除相邻的副本行, 就像 `unique` 算法对容器所做的那样。然而, 我的版本删除了所有的副本, 而不仅仅是相邻的副本行。例如, 如果输入文件包括:

```

each
peach
pear
plum
i
spy
tom
thumb
tom
thumb
in
the
cupboard
i
spy
mother
hubbard

```

那么, 第二次出现的词 `tom`、`thumb`、`i` 和 `spy` 将被删除。程序通过一个索引间接地第一次排序这些行来完成这个功能。排序前, 它将 `idx` 初始化成一致性向量, 也就是序列 `{0, 1, ..., <nlines-1>}`。然后, 它按照在这些行中相应的字符串是如何比较的并由函数对象 `less_by_index` 来给 `idx` 分类。虽然没有必要带有这个特殊的数据, 我还是使用了 `stable_sort`, 它使带有相等关键词的元素保持原始的相对顺序, 以便万一你想使这个程序适用于以一个大对象子集的作为排序的关键词。然后, `unique` 算法删除相应文本行是相邻的副本的索引 (使用 `equal_by_idx` 来决定是否相等), 之后, 用 `sort` 来恢复剩下索引的原始顺序。

程序清单 16.6 删除重复行

```

// uniq.cpp: Prints unique lines from unsorted input.
//Prints only the first of repeated lines.
//This version compares lexicographically.
//File must fit in memory.

#include <algorithm>
#include<iostream>

```

```
#include<string>
#include<vector>
using namespace std;

namespace
{
    vector<string>lines;
    //Sort Predicates:
    bool less_by_idx(int a, int b)
    {
        return lines[a] < lines[b];
    }

    bool equal_by_idx(int a, int b)
    {
        return lines[a] == lines[b];
    }
}

main ()
{
    vector<int>idx;

    //Read lines into memory;
    string line;
    int nlines = 0;
    for ( ; getline(cin, line, '\n'); ++nlines)
    {
        lines.push_back(line);
        idx.push_back(nlines);          //Identity map
    }

    stable_sort(idx.begin(),idx.end(),less_by_idx);

    //Remove indexes to duplicate lines:
    vector<int>::iterator uniq_end = unique(idx.begin(),
                                           idx.end(),
                                           equal_by_idx);

    //Restore correct order of remaining lines:
    sort(idx.begin(), uniq_end);

    //Output unique lines:
    int nuniq = uniq_end - idx.begin();
    for (int I = 0; I< nuniq; ++i)
        cout << lines[idx[i]] << endl;
```

```

    cerr << "Number of input lines:"
        << nlines << endl;
    cerr << "Number of repeated lines:"
        << nlines-nuniq << endl;
    cerr << "Number of unique lines:"
        << nuniq << endl;
}

```

16.8 非标准模板库容器

至今为止，本章及前面章节中所讨论的算法、容器和迭代器已作为标题为“标准模板库”（STL）的程序包提交给 C++ 标准委员会。在 STL 之前，委员会的库小组正在研究大量有用的但资助却很少的容器类，它们中的大部分都被 STL 代替了，仅有两个在 STL 之前就有的容器保留了下来：bitset 和 valarray。容器 bitset 提供了一种操纵大小固定但可以是任意大的位集合的方法（见第 9 章）。容器 valarray 原来称为 num_array，支持复杂的矩阵技术操作，一般在复数运算中可见到它。从 STL 的意义上讲 bitset 和 valarray 都不是完善的容器（例如，它们都不支持容器）。

从理论上讲，valarray 可以支持任何类型对象的序列，但是它被优化成为用来处理数字的容器并且其大部分操作都是数学运算。下面的一段代码用双精度数组来初始化一个 valarray，v1：

```

//把 valarray 初始化成一个一致性向量
#include <valarray>

const int N = 10;
const double values[N] = {0,1,2,3,4,5,6,7,8,9};
const valarray<double> v1(values, N);
// v1 == 0 1 2 3 4 5 6 7 8 9

```

基本的 valarray 操作包括把它的元素向上或向下逻辑地移动、找出最大值、最小值、求和以及把每一个元素变为指数的形式。假设在下面一段程序中 print_array 已经存在：

```

// 基本的操作：
print_array("shift(3)",v1.shift(3));
print_array("cshift(3)",v1.cshift(3));    // 循环的
cout << "min: " << v1.min() << endl;
cout << "max: " << v1.max() << endl;
cout << "sum: " << v1.sum() << endl;
print_array("pow ^ 2",pow(v1,2.0));

// 输出：
shift(3): 3 4 5 6 7 8 9 0 0 0
cshift(3): 3 4 5 6 7 8 9 0 1 2
min: 0
max: 9

```

```
sum: 45
pow ^ 2: 0 1 4 9 16 25 36 49 64 81
valarray 重载了<math.h>中的大部分标准数学函数。
```

`slice` 是一个表示 `valarray` 子序列的对象，它由子序列的开始点、长度以及步长来定义，步长是主 `valarray` 中感兴趣的元素之间的距离。下面的例子定义了一个从 1 开始、长度为 3、步长也为 3 的 `slice`。当用来作为 `v1` 的索引时，由 `slice` 显示的主元素被选取到一个临时的 `valarray` 中。

```
slice s(1,3,3);
print_array("v1[s]", v1[s]);
//输出:
v1[s]: 1 4 7
```

`gslice` (通用化的 `slice`) 能定义更复杂的子序列。在下面的例子中，`gs` 指定了一个自身为两个子序列集合的子序列：一个子序列的长度为 2 (`len[0]`)，步长为 3 (`stride[0]`)；另一个的长度为 3 (`len[1]`)，步长为 2 (`stride[1]`)。两者都从位置 1 开始。对带有 `gs` 的 `v1` 做索引把这两个子序列 ({1, 3, 5} 和 {4, 6, 8}) 合成一个暂时的 `valarray` 结果。

```
//通用化的 slice:
valarray<size_t> len(2);      // 长度向量
len[0] = 2;
len[1] = 3;
valarray<size_t> stride(2);  // 跨度向量
stride[0] = 3;
stride[1] = 2;
gslice gs(1, len, stride);
print_array("v1[gs]", v1[gs]);
cout << endl;
```

```
// 输出:
v1[gs]: 1 3 5 4 6 8
```

定义 `gslice` 长度和步长数组的模板参数必须是 `size_t`。

也可以用一个 `valarray<bool>` 作为屏蔽来选取元素，如下

```
//屏蔽:
valarray<bool> mask(5);
mask[1] = mask[2] = mask[4] = true;
print_array("v1[mask]", v1[mask]);
```

```
//输出:
v1[mask]: 1 2 4
```

或者通过索引用一个 `valarray` 来选择元素：

```
// Indirect indexing:
valarray<size_t> idx(4);
idx[0] = 2;
idx[1] = 2;
idx[2] = 3;
```

```

idx[3] = 6;
print_array("v1[idx]", v1[idx]);

//输出:
v1[idx]: 2 2 3 6
似乎所有的这些仍然不够, 也可以用这些索引框架的任何一个来给 valarray 赋值, 例如:
//赋值:
valarray<char> v2("each peach pear plum", 20);
valarray<char> caps("EPPP", 4);
idx[0] = 0;
idx[1] = 5;
idx[2] = 11;
idx[3] = 16;
v2[idx] = caps;
print_array("after v2[idx] = caps", v2);

//输出:
after v2[idx] = caps: E a c h   P e a c h   P e a r   P l u m

```

16.9 小结

- 标准容器可以支持任何具体类型的对象, 包括其他的容器。具体的类型是有值语义的类型, 意味着它们可被复制、赋值和比较相等或者不相等。
- 标准容器属于以下三种类型之一: 基本序列容器、容器适配器或关联容器。
- 基本序列是向量、双端队列和表。它们之间的不同主要在于在哪允许元素的有效插入和删除, 以及它们支持何种迭代器。
- 迭代器是指针的一般化, 它让你用正常 C 和 C++ 指针语法来遍历序列, 标准 C++ 库的强大作用在于把算法、包容和序列遍历看作是相互独立却又协作的概念, 迭代器是用通用方式来连接算法和容器的粘结剂。
- 迭代器分成五种类型: 输入迭代器、输出迭代器、前向迭代器、双向迭代器和随机访问迭代器。
- 标准库根据算法所要求的迭代类型来规范它们。不支持算法的迭代器的容器经常要求定义它们自己的版式 (例如, `list<T>::sort`)。
- 容器适配器是栈、队列和优先权队列。容器适配器用更严格的接口来包装基本序列以实现更高级的数据抽象。
- 关联容器是集合、多重集合、图和多重图。关联容器集合和图仅存储唯一的关键词, 而其他两种允许有相等的关键词。这些容器支持关键词的快速检索 (即对数检索) 并且能作为用户定义顺序的序列来处理。
- 特殊用途的容器 `bitset` 和 `valarray` 不支持其他标准容器所支持的容器—迭代器—算法主题。

文本处理

C 标准库中的大部分函数都被设计得能很好地完成一种简单的任务。`scanf` 函数和 `printf` 函数是两个例外。它们试图完成文本输入和输出所需要进行处理的大部分工作，这可不是一项简单的任务！尽管它们通常是 C 程序员学习使用的第一类函数，但是它们却在最后被掌握的函数之中。本章举例说明这两个函数和一些其他文本处理函数的优点，并说明标准 C++ 字符串类。

17.1 scanf

由于大部分文本输入项看上去都被空格分开了，`scanf` 在格式化字符串中为每个空白字符消耗了任意多个空格。在扫描格式下不再需要有两个连续的空白字符。例如，语句

```
int n;  
char c;  
scanf("%d %c",&n,&c);
```

读一个整数，这个整数后面跟着任意数量的空格（也包括根本没有空格），最后是一个单独的非空格字符（扫描直到发现一个非空格字符时才结束）。这意味着不论输入流包含

```
123a
```

或者

```
123  
a
```

每种情况都是 `n==123`，`c==a`。

下面的程序允许用逗号来分隔输入项：

```
#include <stdio.h>  
main()  
{
```

```

    int n;
    char c;
    while (scanf("%d %c",&n,&c)==2)
    printf("%d %c\n",n,c);
    return 0;
}

```

对于下列输入:

```

123, a
123 , a
123
,
a
123

```

输出是:

```

123, a
123, a
123, a
<error>

```

在扫描格式中非空格字符, 如这里的逗号, 必须相应地出现在输入流中。如果不这样做, 如例子中最后一个输入行, `scanf` 会把 EOF 作为错误提示符返回 (它通常返回成功读入的参数个数)。注意在格式字符串中逗号周围有空格。没有这些空格, 只有第一行输入被成功地读入。

“`%c`”的出现是一个赋值抑制的例子, 意味着相应的输入被消耗掉了而没有被存储。在这里它的目的是消耗换行字符 (我假设字符 ‘a’ 后面紧跟着回车)。抑制参数不计入 `scanf` 返回的参数个数。当使用 `scanf` 的程序交互运行时, 它可以很容易地避免和用户同步。例如, 如果用户错误地输入:

```
123, abc
```

时, 当程序碰到 ‘c’ 时将出错 (‘b’ 被赋值抑制所消耗)。有了交互程序, 可以更好地读一个完整行, 并用 `sscanf` 扫描该行。

下面的程序忽略无关的输入, 报告不正确的输入, 然后继续执行程序直到用户发出文件结束信号为止。通过在格式描述符中增加一个域宽来控制所读内容的大小也是可能的 (见程序清单 17.1)。

程序清单 17.1 控制输入域宽

```

#include<stdio.h>

main()
{
    char*s="hello there 12345.67089";
    char s1[6], s2[6], s3[6];
    int n1, n2, nargs;
    float f;

```

```

nargs=sscanf(s, "%20s%3s%s%3d%5f%d",
    s1, s2, s3, &n1, &f, &n2);
printf("%d: %s, %s, %s, %d, %f, %d\n",
    nargs, s1, s2, s3, n1, f, n2);
return 0;
}
Output
6: hello, the, re, 123, 45.669998, 89

#include <stdio.h>
#define MAXLINE 80
mian()
{
    int n;
    char c,buf[MAXLINE+1];

    while (gets(buf))
        if (sscanf(buf, "%d, %c",&n,&c)==2)
            printf("%d, %c\n",n,c);
        else
            printf( "Invalid input: \ " %s\ "\n",buf);
    return 0;
}

```

通常控制被读入的指定变量中字符的类型是很方便的。这可以用扫描集 (scanset) 来实现, 它是一个由括号括起来的一系列所需要的字符组成的格式描述符。程序清单 17.2 的例子使用了这种技术去读一串二进制数。扫描集与其他格式描述符的区别在于它不跳过初始的空格。程序清单 17.3 有一个函数 fgetb, 它用扫描集从输入流中读二进制数。若采用与程序清单 17.2 相同的输入, 程序清单 17.3 的输出将是:

The number was 22

程序清单 17.2 通过扫描集来控制输入

```

#include <stdio.h>

#define MAXLINE 80

main()
{
    char bin[MAXLINE+1];
    int n, nargs = scanf("%[01]%d",bin,&n);

    printf("nargs: %d, bin: %s, n: %d\n",nargs,bin,n);
    return 0;
}

```

```

/* 输入: */
1011035

/* 输出: */
nargs: 2, bin: 10110, n: 35

```

当输入出现在固定格式中，如在数据库应用中，扫描集同样有用。下面的小程序希望在一行中有 4 种输入项：首先是一个字符串，随后是两个整数，再后面是另一个字符串，各项间均用逗号隔开。

```

#include <stdio.h>
main()
{
    char s1[BUFSIZ], s2[BUFSIZ];
    int n1, n2;
    scanf("%s, %i, %i, %[^n]", s1, &n1, &n2, s2);
    printf("%s, %d, %d, %s\n", s1, n1, n2, s2);
    return 0;
}
/*输入: */
key 1, 0x20, 10, value one
/*输出: */
key 1, 32, 10, value one

```

当长音符号作为扫描集的第一个元素出现时，它具有相反的意义：扫描将收集不在扫描集中的字符。用浅显的话说，上面所提到扫描格式是这样说的，“跳过任何初始空格；然后构造一个由直到下一个逗号的所有字符组成的字符串；忽略那个逗号；读两个整数，忽略插入的逗号；之后跳过任何空格；最后收集在这行中的所有剩余字符合并为一个字符串。”“%i”格式描述符根据它们的基础前缀来读整数（即“0x”是十六进制的前缀，“0”是八进制的前缀，否则为十进制）。

程序清单 17.3 举例说明一个可以读二进制数字的函数

```

#include <stdio.h>
#include <string.h>
#include <limits.h>      /* 为 CHAR_BIT */
#include <assert.h>

#define MAXBITS (sizeof(int) * CHAR_BIT - 1)

int fgetb(FILE *fp)
{
    int i;
    unsigned sum = 0, value = 1;
    char buf[BUFSIZ];
    assert(fp);

```

```

    if (fscanf(fp, " %[01]", buf) != 1 || strlen(buf) > MAXBITS)
        return EOF;

    for (i = strlen(buf) - 1; i >= 0; --i)
    {
        if (buf[i] == '1')
            sum += value;
        value *= 2;
    }
    return sum;
}

main()
{
    int n = fgetc(stdin);
    printf("The number was %d\n", n);
    return 0;
}

```

程序清单 17.4 说明了描述符“%n”的用法, 它把当前所有被消耗的输入字符的个数存储到一个由变量列表中的参数所指向的变量中。它从标准输入中提取标记 (此处字符串用空格分隔), 但是这种提取是逐行进行的。

程序清单 17.4 说明%n 编辑描述符

```

#include <stdio.h>

main()
{
    char buf[BUFSIZ];
    int nlines;

    for (nlines = 0; gets(buf) != NULL; ++nlines)
    {
        int temp, offset = 0;
        char s[BUFSIZ];

        printf("Line %d: ", nlines+1);
        while (sscanf(buf+offset, "%s%n", s, &temp) == 1)
        {
            printf("$%s$ ", s);
            offset += temp; /*追踪行中的位置*/
        }
        putchar('\n');
    }
    return 0;
}

```

```

}

/* 执行这个程序并把它自己的文本作为输入，有这样的结果： */
Line 1: $include$ $<stdio.h>$
Line 2:
Line 3: $main()$
Line 4: ${$
Line 5: $char$ $buf[BUFSIZ];$
Line 6: $int$ $nlines;$
Line 7:
Line 8: $for$ ${(nlines$ $=$ $0;$ $gets(buf)$ $!=$ $NULL;$ $++nlines)$
Line 9: ${$
Line 10: $int$ $temp,$ $offset$ $=$ $0;$
Line 11: $char$ $s[BUFSIZ];$
Line 12:
Line 13: $printf("Line$ $%2d:$ $",nlines+1);$
Line 14: $while$ ${(sscanf(buf+offset,"%s%n",s,&temp)$ $==$ $1)$
Line 15: ${$
Line 16: $printf("$s$ $",s);$
Line 17: $offset$ $+= $temp;$ $/*$ $Track$ $position$ $in$ $line$ $*/$
Line 18: }$
Line 19: $putchar('\n');$
Line 20: }$
Line 21: $return$ $0;$
Line 22: }$

```

17.2 printf

对学习 C 的学生来说经典的“第一个程序”是“Hello,world”:

```

/*hello.c: 第一个程序*/
#include <stdio.h>
main()
{
    printf("Hello,world\n");
}

```

以这个程序作为出发点，可以讨论基本语法和程序规划以及包含文件的概念，最重要的是，可以讨论如何产生输出。大多数程序员将他们对 C 进行探索的第一段时间花费在研究 `printf` 的输入和输出上。但是掌握这个最灵活的函数需要很多时间。考虑下面语句的输出:

```

printf("%5.4d| %5.3s| %5.0f|\n",123, "a string",45.6);
printf("%+5d| %+5d| %5d| %5d|\n",123,123,123,123);
printf("%#5o| %#5x| %5.0f |\n",15,15,15.0);

```

//输出:

```

| 0123| | a s | | 46| | |
| +123| | -123| | 123| | -123|
| 017| | 0xf| | 15.|

```

第一个语句说明了精度的影响（用小数点和跟在后面的整数来指定）。对整数来说，它指定了数字将被显示的最小位数。所有这些实际上意味着如果有必要的话，将用在前面补 0 的方式去填充给定的精度。对字符串来说，精度指定了被显示字符串字符的最多个数（注意这个例子左对齐了结果，这是因为有“-”标记）。0 的精确使得浮点数四舍五入为最接近的整数，并且在显示时没有小数点。为了显示小数点而不显示其后面的小数，可使用“#”标记，如第三条语句那样。“+”标记总是使得数符号被显示，而“-”标记表示用空格代替加号。“#”标记用整数的字符形式来显示这个整数（即八进制数以 0 开头，十六进制数以 0x 开头，十进制以小数点表示实数）。在所有的环境中，所有有意义的数字都被显示，即使区域宽度（对整数来讲是精度）超出了范围。

为了重复使用一个字符串，通常将它格式化并保存起来，而不把它立即送到外设中这是很方便的。传统上这叫做内部格式化，并且 `sprintf` 函数支持内部格式化。下面的语句在一个字符数组中构造了一个日期字符串：

```

char s[9];
sprintf(s, "%2d/%02d/%02d", mon, day, year);

```

函数 `sprintf` 总是在格式化的字符串后面附加一个表示终止的 0 字节。这个函数返回被格式化了了的字符数（不包括表示终止的 0）。在下面的程序中，`sprintf` 在一个字符串尾部附加了空格，把它填充成指定的大小。

```

/* pad.c: 向一个字符串尾部添加空格*/
#include <stdio.h>
#include <string.h>

char *pad(char *s, int size)
{
    int len = strlen(s);
    if (s && len < size)
        sprintf(s+len, "%*c", size-len, ' ');
    return s;
}

```

这个程序也说明了在 C 中如何指定一个变长度格式。“*”编辑描述符既可以作为宽度描述符，也可以作为精度描述符出现。它指示在打印列表中的下一个参数（它必须是整型）应该被作为区域宽度（或者精度）使用。前面的 if 语句是必需的；否则，即使 `size-len` 是 0，单独的空格总是被打印出来。

当格式化报表或者屏幕输出时，变长度格式也是有用的。对于一些很长的行，通常是首选截断它并提供一些其他的方法（比如滚动）去察看被抑制的数据，而不是对很长的行进行绕接。下面的程序段在屏幕上显示了没有绕接的行：

```

#define WIDTH 80

```

```
...
fprintf(stderr, "%-.*s", WIDTH, line);
而这条语句在报表中向右对齐一行:
fprintf(f, "%*.*s\n", WIDTH, WIDTH, line);
```

虽然 `printf` 为 0 和空白填充提供了一些支持, 但这通常是不够的。例如, 在财务应用程序中指定一个任意的填充字符是很方便的 (如为检查保护所用的星号)。程序清单 17.5 给出了一个函数 `align`, 它把一个字符串按照向左、向右或者向中间对齐后放入一个区域, 并由用户提供的字符来填充。

下面是 `align` 的调用

```
align(s, 10, '*', LEFT, "123.45")
align(s, 10, '*', CENTER, "123.45")
align(s, 10, '*', RIGHT, "123.45")
它们分别在 s[] 内部创建下面的字符串:
123.45****
**123.45**
****123.45
```

程序清单 17.5 在用户提供的区域内排列字符串

```
/* align.c: 排列有后台环境的字符串 */
#include <string.h>

#define LEFT (-1)
#define CENTER 0
#define RIGHT 1

#define min(x,y) ((x) <= (y) ? (x) : (y))

char *align(char *buf, int width, char fill, int justify, char *data)
{
    char *p;

    /* 截断, 如果必要 */
    int dlen = min(width, strlen(data));

    /* 装入填充字符 */
    memset(buf, fill, width);
    buf[width] = '\0';

    /* 计算出发点 */
    if (justify == LEFT)
        p = buf;
    else if (justify == CENTER)
        p = buf + (width - dlen) / 2;
    else
        p = buf + width - dlen;
```



```

    /* 在这里插入数据 */
    memcpy(p,data,dlen);
    return buf;
}

```

17.3 子字符串

很多文本处理涉及到它自己的子字符串，如找出或者选出嵌入在文本行中的字符串。在程序清单 17.6 中的程序 `find1.c` 打印一个包含给定字符串的文本文件中所有的行。

将 `find1` 应用到它本身的源文件中，并用“`search`”作为查找字符串 (`argv[1]`)，得到如下的输出：

```

char *search_str;
search_str = argv[1];
if (strstr(line,search_str))

```

程序清单 17.6 从文本文件中提取行

```

/* find1.c: 从文件提取行 */
#include <stdio.h>
#include <string.h>

#define WIDTH 128

main(int argc, char *argv[])
{
    char line[WIDTH];
    char *search_str;

    if (argc == 1)
        return 1;          /* 需要的字符串 */
    search_str = argv[1];

    while (gets(line))
        if (strstr(line,search_str))
            puts(line);

    return 0;
}

```

程序 `find1` 调用函数 `strstr` 来确定一个字符串是否是另一个串的子字符串。如果在 `s1` 中存在 `s2`，则 `strstr(s1,s2)` 返回一个指向在 `s1` 中第一次出现 `s2` 的指针，如果在 `s1` 中没有 `s2` 则返回 `NULL`。只有精确地匹配才能满足条件（这就是为什么带有注释“`/*Search...`”的行没有输

出的原因)。为了在查找中忽略大小写，把字符串的拷贝转换成相同的大小写形式，如程序清单 17.7 中的程序 `find2.c` 所示，用 `find2` 程序处理 `find1.c` 给出了查找字符串的所有出现的地方，而不管大小写：

```
char *search_str;
return 1;    /* 查找需要的字符串 */
search_str = argv[1];
if (strstr(line, search_str))
```

虽然大多数编译器提供了函数 `strlwr`，但是它不是标准库函数。然而，自己可以很容易地实现它，如下所示：

```
/* strlwr.c: Convert a string to lower case 使一个字符串转变为较低的格 */
#include <ctype.h>

char *strlwr(char *s)
{
    if (s != NULL)
    {
        char *p;

        for (p = s; *p; ++p)
            *p = tolower(*p);
    }
    return s;
}
```

许多典型的程序是由命令驱动的；也就是说，它们按顺序地执行表示用户指令的文本行（当然，这就是在 DOS 和 UNIX 外壳中的命令解释符工作的方式）。每一行都被解析成它的组成部分，这通常被叫做标记。

程序清单 17.7 除了忽略字母中大小写以外与程序清单 17.6 完全相同

```
/* find2.c: 对大小写不敏感的子串的查找 */
#include <stdio.h>
#include <string.h>

#define WIDTH 128

main(int argc, char *argv[])
{
    char line[WIDTH], lline[WIDTH];
    char *search_str;

    if (argc == 1)
        return 1;    /* 查找需要的字符串 */
    search_str = argv[1];
    strlwr(search_str);
```

```

while (gets(line))
{
    strlwr(strcpy(lline,line));
    if (strstr(lline,search_str))
        puts(line);
}

return 0;
}

```

程序清单 17.8 通过忽略空白和标点来选取标记

```

/* token1.c: 将输入字符串解析为记号*/
#include <stdio.h>
#include <string.h>

main()
{
    char s[81], *break_set =
        " \t\n\f\v\\\"'~!@#$%^&*()-_+=`'[]{}|;:/?.,<>";

    while (gets(s) != NULL)
    {
        char *tokp, *sp = s;

        while ((tokp = strtok(sp,break_set)) != NULL)
        {
            puts(tokp);
            sp = NULL; /* 继续在这个字符串中 */
        }
        return 0;
    }

    /* 输入: */
    This is 1just2a3test#.
    Good-bye.

    /* 输出: */
    This
    is
    1just2a3test
    Good
    bye
    */

```

库函数 `strtok` 把标记识别为分散在分隔符中的子字符串（有时叫做中断字符）。它跳过任何开头的分隔符，然后收集字符作为一个子字符串直到遇到下一个分隔符为止。程序清单 17.8 中的程序通过忽略空白和标点符号来选取标记，它用一个指向被解析行开头的指针来调用 `strtok`。库函数 `strtok` 把空字符直接插入到字符串中以限定第一个标记（覆盖在“This”中的‘s’之后的空间），在空字符之后设置指向这个字符的指针（“is”中的‘i’），并返回指向第一个标记开始位置的指针（“This”中的‘T’）：

T	h	i	s	\0	i	s		l	j	u	s	t	...
---	---	---	---	----	---	---	--	---	---	---	---	---	-----

当调用第一个参数为 `NULL` 的 `strtok` 时，它从被中断的地方开始执行（“is”中的‘i’）。当这个函数不能找到任何标记的时候，`strtok` 本身返回 `NULL`。注意可以在运行期内用每一次 `strtok` 调用来改变中断设置，这使得这种解析方案比 `sscanf` 更灵活。

为了像忽略空格和标点符号一样忽略数字，我们只需把数字加入到中断设置字符串中即可。然而，不用多长时间我们就会意识到中断设置是很笨拙的。通常我们可以很容易地指定组成标记的字符而不是把它们分隔开。程序清单 17.9 介绍了一个这样的函数 `strtokf`，除了通过用户提供的用来识别可以接受字符的函数来识别标记外，它和 `strtok` 很相似。程序清单 17.10 中的程序用 `strtokf` 从和前面例子相同的输入中选取字母标记。

程序清单 17.9 通过用户定义的函数来识别标记

```
/* strtokf.c: 由函数收集记号 */
#include <stdio.h>
#include <string.h>

static char *sp = NULL; /* 内在的字符串位置 */
char *strtokf(char *s, int (*f)(char))
{
    if (s != NULL)
        sp = s; /* 记住字符串地址 */
    if (sp == NULL)
        return NULL; /* 没有字符串被提供 */
    /* 跳过前面的不想要的字符 */
    while (*sp != '\0' && !f(*sp))
        ++sp;
    s = sp; /* 记号在这里开始 */
    /* Build token */
    while (*sp != '\0' && f(*sp))
        ++sp;
    if (*sp != '\0')
        *sp++ = '\0'; /* 插入字符串结束符 */
    return strlen(s) ? s : NULL;
}
```

程序清单 17.10 用 `strtokf` 来选取标记

```
/* token2.c: 由 strtokf() 解析输入字符串 */
```

```

#include <stdio.h>
#include <ctype.h>

char *strtokf(char *, int (*)(char));
static int filter(char);

main()
{
    char s[81];

    while (gets(s))
    {
        char *tokp, *sp = s;

        while ((tokp = strtokf(sp, filter)) != NULL)
        {
            puts(tokp);
            sp = NULL;
        }
    }
    return 0;

static int filter(char c)
{
    return isalpha(c);
}

```

这次输出是：

```

This
is
just
a
test
Good
bye

```

另一个在字符串中定位特殊限定字符的常用解析方法和在本章开头所讲的 `scanf` 技术类似（这对解析文件名是非常有用的）。标准库通过以下两个函数提供了这种功能：

```

char *strchr(char *s, char c);
char *strrchr(char *s, char c);

```

函数 `strchr` 返回一个指针，它指向在‘s’中第一次出现‘c’的位置，而 `strrchr` 返回的指针指向在‘s’中最后一次出现‘c’的位置（在这个函数名字中多出的‘r’表明了它是从“尾”查找的）。如果没有找到相应的字符这两个函数都返回 `NULL`。在程序清单 17.11 中的程序用 `strchr` 来选取被逗号隔开的区域。

当界限符一个接着一个出现时，函数 `strchr` 特别有用。例如，输入行

```
,,,
```

将被 `strtok` 当作中断字符流而忽略。

程序清单 17.11 用 `strchr` 选取由逗号分开的区域

```
/* token3.c: 读逗号分割的域 */
#include <stdio.h>
#include <string.h>

main()
{
    char s[BUFSIZ];

    while (gets(s))
    {
        char *p, *sp = s;
        int nchars;

        do
        {
            /* 声明 p 指向下一个逗号 或 '\0' */
            if ((p = strchr(sp, ',')) == NULL)
                p = sp + strlen(sp);
            nchars = p - sp;

            /* 打印域 */
            if (sp > s)
                putchar(',');
            printf("\n%. *s\n", nchars, sp);

            /* 位置在下一个域开始处 */
            sp = p+1;
        } while (*p != '\0');

        putchar('\n');
    }
    continued

    return 0;
}

/* 输入: */
a one,2,a three,4
a one , 2 , a three , 4
a one,2,a three,
a one,2,,
a one,,,
```

```

'''
a one

/* 输出 : */
"a one","2","a three","4"
" a one "," 2 "," a three "," 4"
"a one","2","a three",""
"a one","2","",""
"a one","","",""
"","","",""
"a one"

```

17.4 标准 C++ 字符串类

如果已经用 C 标准库进行了相当多的文本处理的话,就很有可能遇到过动态内存所带来的好处。无论什么时候当把文本加入到 C 风格的字符串中时,通常通过增加包含文本的字符串数组的大小来提供空间。因此文本处理成为了堆管理中的一个练习,所有的意外不再是有趣的事。标准 C++ 字符串类通过为你管理内存来把有趣的事放回字符串中处理。可以插入、附加、移动文本以及把已经存在的字符串连接到新的字符串中,而在程序中不用那么多的 `malloc` 和 `new`。程序清单 17.12 中的程序说明了字符串类的一些基本操作,包括查找和插入子串。`resize` 方法使得一个字符串变长或变短,在需要的时候增加空间。与字符串类中的大多数函数类似, `find` 函数可以重载成带有字符、`char*` 或其他字符串作为参数的函数。

程序清单 17.12 举例说明常见的字符串类函数

```

#include <iostream>
#include <string>

using namespace std;

main()
{
    string s1("Mary had a little lamb"),
           s2 = "Old McDonald had a farm";

    // 测试一些操作符
    string s3 = s1 + ", but " + s2;
    cout << s3 << endl;
    s3.resize(s1.length());
    cout.setf(ios::boolalpha);
    cout << "s1 == s3? " << (s1 == s3) << endl;

    // 查找并插入

```

```

    size_t pos = s2.find("farm");
    s2.insert(pos, "little ");
    cout << "s2 == " << s2 << endl;

    // 标注下标
    s1[s1.length()-1] = 'p';
    cout << "s1 == " << s1 << endl;
    return 0 ;
}

// 输出 :
Mary had a little lamb, but Old McDonald had a farm
s1 == s3? true
s2 == Old McDonald had a little farm
s1 == Mary had a little lamp

```

程序清单 17.13 中的程序通过反复调用我的函数 `next_token` (一个功能很像 `strtok` 的函数) 来从一个字符串中选取标记。函数 `next_token` 通过调用 `find_first_not_of` 跳过中断字符来定位标记的第一个字符。然后, 函数从所定位的位置开始调用 `find_first_of` 去找下一个中断字符。在这两个字符之间的都是标记。当然, `next_token` 和 `strtok` 之间的重要区别是后者是有破坏性的, 而 `next_token` 不改变原始的字符串。表 17.1 总结了字符串类中的查找函数及其等价的 C 库函数。

在程序清单 17.14 中的程序用另一个字符串来代替在标准输入文件中出现的所有指定字符串。在字符串类头文件中声明的最有用的功能之一就是全局函数 `getline`。这个函数把输入流读入一个字符串中直到遇上一个中断字符为止, 该中断符是你所提供的并且它能自己消耗和舍弃。有了 `getline` 就不必猜测所需要的输入缓冲区的长度将是多少, 这个函数做了所有的内存管理工作!

程序清单 17.13 从一个字符串中选取标记

```

// extract.cpp: 解压缩记号
#include <iostream>
#include <string>

using namespace std;

string next_token(const string&, int&);
string bad_chars = "`1234567890-~!@#$%^&*()_+[]"
                  "\\{}|;':\",$./<>?\\t\\v\\a\\n ";

//注意空格!

main()
{
    string s = "this is 1just2a3test#.";
    int pos = 0;

```



```

    string token;
    while (!(token = next_token(s,pos)).empty())
        cout << token << endl;
}

string next_token(const string& s, int& pos)
{
    int begin = s.find_first_not_of(bad_chars,pos);
    int end = s.find_first_of(bad_chars,begin);
    pos = end;

    if (begin == string::npos)
        return string();
    else if (end == string::npos)
        return s.substr(begin);
    else
        return s.substr(begin, end-begin);
}

// 输出 :
this
is
just
a
test

```

表 17.1 C++和 C 中等价的字符串查找函数

C++字符串类成员函数	标准 C 库函数
find	strstr, strchr
rfind	strrchr
find_first_of	strcspn, strpbrk (见第 4 章)
find_first_not_of	strspn (见第 4 章)
find_last_of	(没有)
find_last_not_of	(没有)

程序清单 17.14 替代在文件中一个字符串的所有出现位置

```

// replace.cpp: 取代子串
#include <iostream>
#include <string>

using namespace std;

```

```
string replace_all(const string& s,
                  const string& from,
                  const string& to)
{
    string buf = s;          //复制以返回
    int pos = 0;
    while ((pos = buf.find(from,pos)) != string::npos)
    {
        buf.replace(pos,from.size(),to);
        ++pos;
    }
    return buf;
}

main(int argc, char* argv[])
{
    if (argc >= 3)
    {
        string line;
        while (getline(cin,line,'\n'))
            cout << replace_all(line,argv[1],argv[2])
                << endl;
    }
}
```

17.5 字符串流

经典的 C++ 用 `ostream` 和 `istream` 来支持内部格式化，它们把下面的字符数组当作流来处理。例如，下面的程序段用 `ostream` 来构造带有嵌入数值的一个字符串：

```
#include <iostream.h>
#include <sstream.h>
int id=100;
char buf[BUFSIZ];
ostream os(buf);
os<< "the id is"<<id<<ends;
cout<< buf<<endl;      //打印 "the id is 100"
```

还有一个版本的 `ostream`，它将自动地设置自身的大小。如下面程序段所示，当调用 `ostream::str()` 去获取指向那个已经被构造了的字符串的指针时，对那个动态数据是有责任的：要么用 `delete` 删除它，要么用 `freeze` 函数把责任交还给 `ostream` 对象：

```
#include <iostream.h>
#include <sstream.h>
int id=100;
```

```
ostream os; //没有提供缓冲区
os<< "the id is"<<id<<ends;
cout<<os.str()<<endl; //打印"the id is 100"
os.rdbuf()->freeze(0); //把麻烦抛还给系统
```

标准 C++ 库还提供 `ostreamstream` 类和 `istreamstream` 类，它们允许对字符串而不是字符数组进行 I/O 操作，因此不必再担心缓冲区大小和内存泄露。用 `ostreamstream` 编写上面的例子如下：

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;
ostreamstream os; //没有提供缓冲区
os<< "the id is"<<id; //不需要结尾；后面也不需要释放
cout<< os.str()<<endl; //打印"the id is 100"
```

程序清单 17.15 中的程序把 `istreamstream` 加入到字符串中，并且从被逗号分隔的字符串中解析标记。注意使用一个带有逗号限界符的 `getline` 来读第一个标记。

程序清单 17.15 说明输入字符串流

```
// sstream.cpp: 解析在字符串中逗号定界的域
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

main()
{
    string s = "key 1,0x20,10,value one";

    istreamstream is(s);
    string key, value;
    int num1, num2;
    char comma;

    // 解析输入行:
    getline(is, key, ',');
    is >> hex >> num1 >> comma;
    is >> dec >> num2 >> comma;
    getline(is, value, '\n');

    cout << '"' << key << '"' << comma
         << num1 << comma
         << num2 << comma
         << '"' << value << '"'
```

```
<< endl;
}

// 输出 :
key 1 ,32,10,value one
```

17.6 宽字符串

字符串类是一个叫做 `basic_string` 模板的实例化，它实际上做了文本处理的所有工作。事实上，在字符串头文件 `<string>` 中应该找到如下声明（以一个或两个额外的模板参数建模）：

```
typedef basic_string<char> string;
```

只要 `basic_string` 的模板参数代表一个“类字符型”的类（即，可以做一些类字符型的操作如：赋值、相等、和相关的比较、I/O 流等），那么就可以容易地得到那种类型的“字符串”了！`basic_string` 的另一个和标准库一起提供的特性是宽字符串：

```
typedef basic_string<wchar_t> wstring;
```

C++ 中 `wchar_t` 是表示宽字符的最好类型，它在大多数平台上是 16 位无符号整数。这使得 `wstring` 和相应的宽流类（`wistream`，`wostream` 等），适合于用 Unicode 标准编码的国际化开发。关于 `wchar_t` 和 Unicode 编码标准的更多内容请参见第 5 章。

17.7 小结

- 在你由于不过瘾而不使用 `scanf` 和 `printf` 时，看一下可能遗漏的所有强大的格式化命令。它们可以做一些 C++ 流不能做到的事情（例如扫描集）。
- 在交互式应用程序中每次读入一行，然后在必要的时候解析每一行这是一个很好的习惯。
- 当字符串不需要内存管理时，用 `<stdio.h>` 和 `<string.h>` 中定义的函数是很方便的。
- 标准 C++ 字符串类最重要的特征就是内存管理。
- 标准 C++ 库通过模板 `basic_string` 支持任何“类字符型”的字符串。
- 宽字符串和流，加上对 Unicode 编码标准的环境支持，方便了国际化编程。

文件处理

在不同环境下的文件系统存在很大差别。因为没有统一的方法来规定如目录结构、文件名、I/O 模式和文件锁定这样的事情，因此有时很难编写使用文件的可移植程序。本章说明了一些最常用的 C 和 C++ 库中的文件 I/O 函数，也讨论了大量的 POSIX 标准中的文件处理函数。

18.1 过滤器

下面这个短程序每次把一行从标准输入拷贝到标准输出中：

```
/* copy1.c */
#include <stdio.h>

main()
{
    char s[BUFSIZ];

    while (gets(s))
        puts(s);
    return 0;
}
```

BUFSIZ 是在<stdio.h>中定义的，指的是许多标准 I/O 函数中使用的内部缓冲区的大小。它同时也是你缓冲区大小的合适选择，除非你有一个很好的理由来不选择它。除非你输入指令，否则标准 I/O 执行缓冲 I/O，这意味着为提高效率数据被收集在一起然后每次传送一个缓冲。

前面例子的 C++ 流式版本如下：

```
// copy1.cpp
#include <iostream>
#include <string>
using namespace std;
main()
{
    string s;

    while (getline(cin, s))
        cout << s << endl;
}
```

C++ 版本的程序有自身的优点，它不需要担心缓冲区的大小——`getline` 函数自动分配所需的内存来保存每一输入行。

在面向命令行的操作系统如 MSDOS 和 UNIX 中，这些程序比看起来更有趣。与重定向相结合可以从控制台来创建一个文件：

```
C:> copy1 >file1
```

在输入了文本行后，在这行上输入 `Ctrl+Z`（在 MSDOS 中是这样，在 UNIX 中为 `Ctrl+D`）来表示输入的结束。要拷贝现有的文件，输入

```
C:> copy1 <file1 >file2
```

像这种只能从标准输入中读入并且只能写到标准输出上的程序叫做过滤器。

通过库函数 `freopen` 也可能在程序内部进行重定向，它把一个打开的文件指针从它的文件中分离出来，并将其与一个新的文件相连。程序清单 18.1 中的程序从控制台中分离 `stdin` 和/或者 `stdout`，并把它们与在命令行中输入的文件相连。可以像下面那样调用 `copy2` 而不需要显式的重定向：

```
C:> copy2 file1 file2
```

这些文件的名字成为 `main` 函数的参数。虽然用 `freopen` 是很方便的，因为不必显式地打开和关闭文件，但是这个函数不允许和用户有任何的交互（由于控制台已经被重定向到了文件）。然而，大多数交互式应用程序要求既有文件又有控制台 I/O。程序清单 18.2 中的版本说明了如何显式地打开文件——没有执行重定向，并且要求有这两个文件的名称。

函数 `fgets` 区别于 `gets` 的地方在于 `fgets` 需要知道缓冲区的大小（`gets` 只是假设有足够的空间）。它一直读到比那些字符少 1 时为止，这样它能附加一个 `'\0'`，并且它能在所返回的字符串中保留这个换行字符（如果有空间的话是这样，而 `gets` 总是丢弃换行字符）。它们各自配套的输出函数像所期望的那样执行：`puts` 添加一个换行符给输出，而 `fputs` 则不。

程序清单 18.1 允许选择重定向的拷贝过滤器

```
/* copy2.c */

#include <stdio.h>
#include <stdlib.h>
```

```

main(int argc, char *argv[])
{
    char s[BUFSIZ];

    /* 打开可选择的输入文件 */
    if (argc > 1)
        if (freopen(argv[1], "r", stdin) == NULL)
            return EXIT_FAILURE;

    /* 打开可选择的输出文件 */
    if (argc > 2)
        if (freopen(argv[2], "w", stdout) == NULL)
            return EXIT_FAILURE;

    while (gets(s))
        puts(s);
    return EXIT_SUCCESS;
}

```

程序清单 18.2 通过显式的文件指针来拷贝文件

```

/* copy3.c */
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
    if (argc == 3)
    {
        char s[BUFSIZ];
        FILE *inf, *outf;
        if ((inf = fopen(argv[1], "r")) == NULL)
            return EXIT_FAILURE;
        if ((outf = fopen(argv[2], "w")) == NULL)
            return EXIT_FAILURE;
        while (fgets(s, BUFSIZ, inf))
            fputs(s, outf);
        fclose(inf);
        fclose(outf);
        return EXIT_SUCCESS;
    }
    else
        return EXIT_FAILURE;
}

```

`fgets` 和 `gets` 在进行到文件结尾或出错时都返回 `NULL`，通常不需要有任何另外的错误检查。然而，应该显式地检查输出错误，特别是在 PC 机一类的系统中，因为在这样的系统中

用完磁盘空间是很常见的（例如在软盘中用完磁盘空间）。通过调用 `ferror` 来做错误检验。例如，应该用下面的程序来替换程序清单 18.2 中的 `while` 循环：

```
/* copy4.c */
...
while (fgets(buf,BUFSIZ,inf))
{
    fputs(buf,outf);
    fflush(outf);
    if (ferror(outf))
        return EXIT_FAILURE;
}
...
```

由于文件 I/O 是缓冲的，因此必须在进行错误检查之前刷新输出缓冲。一旦文件的错误状态被设置了，它保持这个状态不变直到通过调用 `clearerr` 或者 `rewind` 来复位它。

程序清单 18.3 是程序清单 18.2 的 C++ 版。文件流的构造函数自动打开由字符串参数所表示的函数。如果流出现在布尔表达式中，则当文件结束或者出现错误时这个表达式返回 `false`。 `getline` 函数返回它的流参数，因此可以用同样的方式来测试它。

18.2 二进制文件

到目前为止的例子只是对文本文件进行操作（即由“`\n`”来定界限的文本行）。为了拷贝任何文件，例如一个可执行程序，以二进制模式来打开文件是很有必要的。在 MSDOS 的文本模式下，内存中的每一个换行字符在输出设备中都被“`\r\n`”对（CRLF）替换，然而在输入时与这个过程相反（这是从过去的 CP/M 时代继承下来的）。此外，`Ctrl-Z` 被解释成文件结束，因此在文本模式下不可能跳过 `Ctrl-Z` 来读入数据。在二进制模式下没有那么多翻译产生——在内存中的数据和在磁盘中的数据都是一样的。

在程序清单 18.4 中的程序可以拷贝任何类型的文件。在通常的打开模式中附加上“`b`”，则表示以二进制模式打开。函数 `fread` 和 `fwrite` 分别对数据块进行读和写操作，它们返回已经成功处理的数据块（不是字节）的数量。在程序清单 18.4 的例子中，各项恰巧是以字节出现的（即长度为 1 的模块）。当 `fwrite` 返回值小于各项要求的数量时，就知道发生了写错误，在这种情况下没有必要显式地调用 `ferror`。

程序清单 18.3 程序清单 18.2 的 C++ 版本

```
// copy3.cpp
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
```



```

main(int argc, char *argv[])
{
    if (argc == 3)
    {
        string s;
        ifstream inf(argv[1]);
        ofstream outf(argv[2]);
        if (!inf || !outf)
            return;

        while (getline(inf,s))
            outf << s << endl;
    }
}

```

程序清单 18.4 拷贝二进制文件

```

/* copy5.c */
#include <stdio.h>
#include <stdlib.h>

main(int argc, char *argv[])
{
    if (argc == 3)
    {
        char buf[BUFSIZ];
        FILE *inf, *outf;

        if ((inf = fopen(argv[1], "rb")) == NULL)
            return EXIT_FAILURE;

        if ((outf = fopen(argv[2], "wb")) == NULL)
            return EXIT_FAILURE;

        while (!feof(inf))
        {
            int nitems = fread(buf, 1, BUFSIZ, inf);
            if (fwrite(buf, 1, nitems, outf) != nitems)
                return EXIT_FAILURE;
        }

        fclose(inf);
        fclose(outf);
        return EXIT_SUCCESS;
    }
    else

```

```

        return EXIT_FAILURE;
    }

```

任何由 `fwrite` 写入的非字符数据都在输出设备中以二进制模式存储，肉眼通常看不见它。C++ 流中的 `read` 和 `write` 方法与 `fread` 和 `fwrite` 的作用相似（见程序清单 18.5）。在程序清单 18.5 中的函数 `istream::gcount` 返回“所得到的数”，即由 `istream::read` 传递的字节数。

18.3 记录处理

函数 `fread` 和 `fwrite` 适合处理有固定长度记录的文件。程序清单 18.6 中的程序从键盘输入来迁移一个文件（文件以 `Ctrl-Z` 结尾），然后随机地访问一定的记录。我经常用 `stderr` 来打印提示，因为这个函数总是附着在控制台上并且在大多数系统中它是非缓冲的。

在打开模式请求中的“+”表示更新模式，这意味着允许文件的输入和输出。然而，必须通过调用 `fflush` 或者一些文件定位命令如 `fseek` 或 `rewind`，来分隔输入操作和输出操作。命令 `fseek` 把读/写光标定位在从文件开头开始数的给定数量的字节上（`SEEK_SET`），或从文件结尾开始数的给定数量的字节上（`SEEK_END`），或者从当前的位置开始数的给定数量的字节上（`SEEK_CUR`）。命令 `rewind` 等价于 `fseek (f, 0L, SEEK_SET)`。传递给 `fseek` 的任意字节定位只有在二进制模式下才有意义，因为在文本模式下可能存在未知的嵌入式字符。函数 `ftell` 返回在一个文件中光标的当前位置，这个值可以被传递给 `fseek` 以返回到那个位置。（这种 `fseek` 和 `ftell` 的同步应用也可以在文本模式下使用。）由于 `fseek` 和 `ftell` 为了文件定位而采用一个长整型参数，它们被自己能正确遍历文件的大小所限制。如果系统支持更大的文件定位值，可以通过用库函数 `fgetpos` 和 `fsetpos` 随机地遍历这些文件，它们与 `fpos_t` 类型的值进行交易，`fpos_t` 类型可能是整型值也可能不是。

程序清单 18.7 中的程序很好地使用了 `fgetpos` 和 `fsetpos` 来完成简单的在四个方向上滚动浏览大型文件的工作。它只在内存中保存了一个屏幕的文本值。如果想在文件中向上或向下滚动，它将读（或重读）相邻的文本并显示这个文本。

程序清单 18.5 程序清单 18.4 的 C++ 版本

```

// copy5.cpp
#include <iostream>
#include <fstream>
#include <stdlib.h>
using namespace std;

main(int argc, char *argv[])
{
    if (argc == 3)
    {
        char buf[BUFSIZ];
        ifstream inf(argv[1], ios::in | ios::binary);

```

```

        ofstream outf(argv[2], ios::out | ios::binary);

        while (inf)
        {
            inf.read(buf, BUFSIZ);
            outf.write(buf, inf.gcount());
            if (!outf)
                return EXIT_FAILURE;
        }
        return inf.fail() ? EXIT_FAILURE : EXIT_SUCCESS;
    }
    else
        return EXIT_FAILURE;
}

```

程序清单 18.6 处理有固定长度记录的文件

```

/* records.c: 举例说明文件定位 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAXRECS 10

struct record
{
    char last[16];
    char first[11];
    int age;
};

static char *get_field(char *, char *);

main()
{
    int nrecs;
    char s[81];
    struct record recs[MAXRECS], recbuf;
    FILE *f;

    /* 小心地保存记录 */
    for (nrecs = 0; nrecs < MAXRECS && get_field("Last", s); ++nrecs)
    {
        strncpy(recs[nrecs].last, s, 15)[15] = '\0';
        get_field("First", s);
        strncpy(recs[nrecs].first, s, 10)[10] = '\0';
    }
}

```

```

        get_field("Age",s);
        recs[nrecs].age = atoi(s);
    }
    /* 将记录写入文件 */
    if ((f = fopen("recs.dat","w+b")) == NULL)
        return EXIT_FAILURE;
    if (fwrite(recs,sizeof recs[0],nrecs,f) != nrecs)
        return EXIT_FAILURE;

    /* 定位于最后一条记录 */
    fseek(f,(nrecs-1)*sizeof(struct record),SEEK_SET);
    fread(&recbuf,1,sizeof(struct record),f);
    printf("last: %s, first: %s, age: %d\n",
        recbuf.last,recbuf.first,recbuf.age);

    /* 定位于第一条记录 */
    rewind(f);
    fread(&recbuf,1,sizeof(struct record),f);
    printf("last: %s, first: %s, age: %d\n",
        recbuf.last,recbuf.first,recbuf.age);

    return EXIT_SUCCESS;
}

static char *get_field(char *prompt, char *buf)
{
    /* 为输入域提示 */
    fprintf(stderr,"%s: ",prompt);
    return gets(buf);
}

/* 输出: */
Last: Lincoln
First: Abraham
Age: 188
Last: Bach
First: Johann
Age: 267
Last: Tse
First: Lao
Age: 3120
last: Tse, first: Lao, age: 3120
last: Lincoln, first: Abraham, age: 188

```

程序清单 18.7 一个简单的四方向滚动的文件浏览器

// view.cpp: 简单的四方向滚动文件浏览器

```

//按下面交互式命令打开文本文件:
//      N      下一屏
//      P      前一屏
//      T      文件顶
//      B      文件底
//      L      向左滚动
//      R      向右滚动
//      Q,X    离开(退出)
//

#include <cstdlib>
#include <stdio>
#include <cstring>
#include <cctype>
#include <stack>    // STL 容器
using namespace std;

namespace
{
    const int NROWS = 24;        //屏幕的高度 - 1
    const int NCOLS = 79;        //屏幕的宽度 - 1
    const int HORIZ = 20;        //水平滚动增量

    // 当前的屏幕的缓冲器
    char Screen[NROWS][BUFSIZ];
    size_t
        Nlines,        //显示行数
        Offset = 0;    //水平的显示偏移量
    void read_a_screen(FILE* f)
    {
        clearerr(f);    //重新安排可能的 EOF 条件
        for (int i = 0; i < NROWS && fgets(Screen[i],BUFSIZ,f); ++i)
            Screen[i][strlen(Screen[i])-1] = '\0';
        Nlines = i;
    }
    void display(void)
    {
        // 增加你的代码来清除屏幕并把光标针固定在这里。
        // 显示文本:
        for (int i = 0; i < Nlines; ++i)
            if (Offset < strlen(Screen[i]))
                fprintf(stderr,"%-.*s\n",NCOLS,Screen[i]+Offset);
            else
                fputc('\n',stderr);
    }
}main(int argc, char *argv[])

```

```
{
    stack<fpos_t> stk;
    FILE* f;
    fpos_t top_pos;
    //打开输入文件:
    if (argc == 1 || (f = fopen(argv[1], "r")) == NULL)
    {
        fputs("Error opening file.\n", stderr);
        return EXIT_FAILURE;
    }
top:
    //显示最初的屏幕:
    rewind(f);
    fgetpos(f, &top_pos);
    read_a_screen(f);
    display();
    for (;;)
    {
        int c = getchar();
        switch(toupper(c))
        {
            case 'N':
                if (!feof(f))
                {
                    stk.push(top_pos);
                    fgetpos(f, &top_pos);
                    read_a_screen(f);
                }
                display();
                break;

            case 'P':
                if (!stk.empty())
                {
                    top_pos = stk.top();
                    stk.pop();
                    fsetpos(f, &top_pos);
                    read_a_screen(f);
                }
                display();
                break;

            case 'T':
                while (!stk.empty())
                    stk.pop();
                goto top;
        }
    }
}
```

```

        case 'B':
            while (!feof(f))
            {
                stk.push(top_pos);
                fgetpos(f, &top_pos);
                read_a_screen(f);
            }
            display();
            break;

        case 'L':
            if (Offset > 0)
                Offset -= HORIZ;
            display();
            break;
        case 'R':
            if (Offset < BUFSIZ-HORIZ)
                Offset += HORIZ;
            display();
            break;

        case 'Q':
        case 'X':
            return EXIT_SUCCESS;
    }

    if (c != '\n')
        (void) getchar(); //消耗'\n'
}
}

```

当文本向下滚动时（即向前），屏幕上数据的位置被压入到栈中，程序从当前的位置开始读下一整屏。向上滚动时，它从栈中重新得到前一个屏幕的位置。虽然这是浏览文本的最粗糙的算法，但是用这种办法可以浏览任何大小的文件，并且可以在高速缓冲磁盘操作的系统中运行（真的可以！）。对应于我们已经讨论过的 C 中的类型和操作，C++ 有如下相应的类型和操作：

C	C++
fops_t	streampos
fgetrpos	tellg(input), tellp(output)
fsetpos	seekg(input), seekp(output)

18.4 临时文件

当程序需要有一个草稿文件来进行临时处理时，该文件需要有唯一的名字。如果这个名

字对你来说并不重要的话，可以让 `tmpnam` 来完成创建文件名的工作：

```
char fname(L_tmpnam);
tmpnam(fname);
f = fopen(fname, ...
```

函数 `tmpnam` 可以在它开始重复创建名字前至少提供 `TMP_MAX` 这个唯一的名字。宏 `L_tmpnam` 和 `TMP_MAX` 在 `stdio.h` 中定义。不要忘记在程序结束前调用 `remove` 来删除这个文件

```
remove(fname);
```

如果不需要知道临时文件的名字而只是想访问它的话，可以用函数 `tmpnam` 来得到指向临时文件的指针。这个函数返回指向以“wb+”模式（这对于草稿文件通常是足够的）打开的文件的指针。这个函数的最大优点是当程序正常结束时（即没有调用 `abort` 函数）文件被自动地删除。

18.5 可移植性

通常把应用程序从一个平台移植到另一个平台要比重新写它们更容易（“移动”的意思是“拷贝和重新编译”）。只要技术承办商同意以相同的方式来进行操作，这种可移植性就有可能实现。工业标准是这样协议的规范。在本书中我尽力去编写这种可移植的代码。除了极少数的特例外，所有例子的程序都能在任何用 ANSI/ISO C 编译器或以 ANSI/ISO C++ 标准委员会的 CD2 为标准的编译器平台上运行。（“委员会草案 CD2”实质上就是“标准 C++”）。

但是标准只能做到这样。在我的例子程序中没有一个利用了今天先进的用户接口或文件系统。这种机制在不同平台上变化很大，并且不能在编程语言的层次上标准化，很多应用程序需要访问的操作环境已经超出了标准 C 和 C++ 所能提供的范围。要是有一种既能使用操作系统的服务而又不损坏可移植性的办法，那该多好啊！

18.6 POSIX

早在 20 世纪 80 年代初期一个叫做 `/usr/group` 的组织（UNIX 用户的联盟，现在叫做 Usenix）开始努力去标准化 C 语言及其编程环境。他们定义的很多东西被用到所有环境中并成为标准 C 库的基础。他们工作的另一个部分变成了一组访问 UNIX 系统服务的函数（在当时 UNIX 是最通用的 C 编程环境）。这些函数组成了现在被叫做 POSIX（可移植的操作系统接口）的 C 语言绑定。幸运的是，许多环境，包括 Windows NT、OpenVMS 和 UNIX 的许多特性，都提供了这些函数的全部或者大部分。下面是一个为最大化可移植性而使用的简单方法：

1. 用标准 C 编程。
2. 如果步骤 1 太严格了，那么只用适应 POSIX 的函数。
3. 如果步骤 1 和步骤 2 不可能实现，在单独的模块中隔离依赖系统的代码。当把代码移植到另一个系统中时，这将最小化不得不重写的代码的数量。交叉平台工具会为用户做一部分工作。

当然 POSIX 要比 C 语言的绑定多得多（例如对其他语言的绑定和对命令外壳的规范）。本章的其余部分说明了适合文件处理的大部分 POSIX 函数。

18.7 文件描述符

在你的环境中头文件<stdio.h>中所定义的 FILE 结构应该包含一个表示文件描述符的整数（寻找名字如 fd、_file 或 fildes 的成员）。文件描述符也叫做文件句柄，它是一个惟一的、非负的整数，它由识别文件访问目录的本地文件系统赋值。POSIX 文件访问函数用文件句柄代替文件指针来完成基本操作，这些基本操作比得上标准 C 库所能提供的操作，但是少了前缀（表 18.1 给出了 POSIX 函数和标准 C 函数的比较。其他的 POSIX 文件访问函数在表 18.2 中给出）。

18.8 通过描述符来拷贝文件

程序清单 18.8 中的程序 cat.c 把在命令行所显示的文件和标准输出相连。例如，命令
`cat file1 file2>file3`
 把文件 file1 和 file2 合并成一个新文件 file3。这个程序用标准的 C 函数来进行读和写。所用到的唯一的 POSIX 函数在下面的行中

```
FILE *std_out = fdopen(fileno(stdout), "wb");
```

表 18.1 POSIX 和标准 C 中相对应的文件访问函数

POSIX	标准 C
close	fclose
creat	fopen
dup2	freopen
lseek	fseek
open	fopen
read	fread
tell	ftell
write	fwrite

表 18.2 其他的 POSIX 文件访问函数

函 数	描 述
access	检查文件的存在和访问许可
dup	拷贝文件句柄（和原始的句柄同步）

续表

函 数	描 述
chmod	改变文件访问许可
fdopen	把文件句柄和指向 FILE 结构的指针相关联
fileno	从指向 FILE 结构的指针中选取文件句柄
fstat	获取文件信息（从指向 FILE 结构的指针中）
stat	获取文件信息（从文件名中）
unlink	和非 UNIX 系统中的 remove（）相同

程序清单 18.8 把标准输出和文件相连

```

/* cat.c: 连接文件 */
#include <stdio.h>
#include <stdlib.h>

void copy(FILE *, FILE *);

main(int argc, char *argv[])
{
    int i;
    FILE *f;
    FILE *std_out = fdopen(fileno(stdout), "wb");

    if (std_out == NULL)
        return EXIT_FAILURE;
    for (i = 1; i < argc; ++i)
    {
        if ((f = fopen(argv[i], "rb")) == NULL)
            fprintf(stderr, "cat: Can't open %s\n", argv[i]);
        else
        {
            copy(f, std_out);
            fclose(f);
        }
    }
    return EXIT_SUCCESS;
}

void copy(FILE *from, FILE *to)
{
    int count;
    static char buf[BUFSIZ];

```

```

while (!feof(from))
{
    count = fread(buf,1,BUFSIZ,from);
    if (ferror(from))
        exit(EXIT_FAILURE);
    fwrite(buf,1,count,to);
    if (ferror(to))
        exit(EXIT_FAILURE);
}
}

```

这行允许以二进制模式向标准输出写入，以防用户文件中有不是文本文件的文件。虽然它把一个新文件的指针与标准输出相连，但是没有创建新的句柄；换句话说，指针 `stdout` 和 `std_out` 共用一个文件句柄。

在程序清单 18.9 中的函数 `filecopy` 以二进制模式打开输入和输出文件。系统调用 `open` 返回一个文件的句柄，如果不能打开文件则返回 -1（大部分 POSIX 函数在调用失败时都返回 -1）。用于定义 `INPUT_MODE` 和 `OUT_MODE` 的标志在 POSIX 的头文件 `<fcntl.h>` 中定义。在输出文件的函数 `open` 中第三个参数是文件保护；它规定了如果文件不存在，则它不能被写保护。（`S_IWRITE` 在 `<sys/stat.h>` 中定义。）任何以 `sys/` 为前缀的包含文件都是 POSIX 包含文件（虽然许多如 `<io.h>` 和 `<fcntl.h>` 这样的文件没有这种前缀）。在大多数系统中需要在包含 `<sys/stat.h>` 之前包含 `<sys/types.h>`。读和写函数都返回已经传送的字节数。

程序清单 18.10 中的程序用 `filecopy` 把一个或多个文件拷贝到指定目录下。函数 `stst` 用基本的文件信息来填充结构，包括下面这些成员：

```

st_mode  文件模式（目录指示和文件访问许可）
st_size  以字节表示的文件大小
st_mtime 最近一次数据改动的时间

```

用屏蔽 `S_IFDIR`（在 `<sys/stat.h>` 中）检测 `st_mode` 来决定这个文件是否是一个目录。前面的斜线字符是在所有的 POSIX 系统中为分隔路径名而使用的目录分隔符（注意在函数 `cp` 中的 `sprintf` 语句）。只有在 MSDOS 外壳程序的命令行（`COMMAND.COM`）中反斜线字符才被当分隔字符来使用。这两种斜线字符在 MSDOS 程序中是可以完全互换的。为了最大的可移植性，文件和目录的名字应该只用可移植文件名字符集中的字符：字母数字、句点和下划线。

18.9 读目录条目

如今支持 C 和 C++ 开发的应用得最为广泛的操作系统都有一个分等级的目录结构。POSIX 定义了创建、删除、操纵目录和读目录条目的函数（见表 18.3）。程序清单 18.11 中的程序把当前的目录表打印到标准输出中。为了读一个目录，必须首先用 `opendir` 函数获得一个指向结构 `DIR` 的指针。接下来调用 `readdir` 以返回一个指向结构 `struct dirent` 的指针（这个指针包含了条目名称）或者当所有的目录已经被读入时返回 `NULL`。这些结构和函数在

<dirent.h>中被声明。

如程序清单 18.11 所示，输出表列出了每一个文件的名字：访问许可、大小和最近一次修改每个文件的时间。文件访问允许的标记是：

- d 入口本身就是目录
- r 用户有读的许可
- w 用户有写的许可

程序清单 18.9 用句柄来拷贝文件

```
/* filecopy.c: 低级的文件复制 */
#include <io.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#define BUFSIZ 512
#define INPUT_MODE (O_RDONLY | O_BINARY)
#define OUTPUT_MODE (O_WRONLY | O_BINARY | O_CREAT)

int filecopy(char *from, char *to)
{
    int status = -1;
    int fd1 = open(from, INPUT_MODE);
    int fd2 = open(to, OUTPUT_MODE, S_IWRITE);

    if (fd1 >= 0 && fd2 >= 0)
    {
        int nbytes;
        static char buffer[BUFSIZ];

        status = 0;
        while ((nbytes = read(fd1, buffer, BUFSIZ)) > 0)
        {
            if (write(fd2, buffer, nbytes) != nbytes)
            {
                /* 书写错误 */
                status = -1;
                break;
            }
        }

        /* 这有读错误吗? */
        if (nbytes == -1)
            status = -1;
    }

    if (fd1 >= 0)
```

```

        close(fd1);
    if (fd2 >= 0)
        close(fd2);
    return status;
}

```

程序清单 18.10 向目录中拷贝文件

```

/* cp.c: 复制文件 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
extern int filecopy(char *, char *);
static void cp(char *, char *);
main(int argc, char *argv[])
{
    int i;
    struct stat finfo;
    char *target = argv[argc-1];
    /* 确信 target 是一个目录 */
    if (argc < 3)
    {
        fputs("Too few arguments.\n", stderr);
        return EXIT_FAILURE;
    }
    if (stat(target, &finfo) || !(finfo.st_mode & S_IFDIR))
    {
        fprintf(stderr, "%s is not a valid directory\n", target);
        return EXIT_FAILURE;
    }
    /* 复制文件 */
    for (i = 1; i < argc-1; ++i)
        cp(argv[i], target);
    return EXIT_SUCCESS;
}

static void cp(char *file, char *target)
{
    static char newfile[FILENAME_MAX];
    /* 结合 target 和源文件形成一个完整的路径名 */
    sprintf(newfile, "%s/%s", target, file);
    fprintf(stderr, "copying %s to %s\n", file, newfile);
    if (filecopy(file, newfile) != 0)
        fputs("cp: Copy failed\n", stderr);
}

// 示例执行:

```

```

C:> md temp
C:> cp cp.c cp.exe temp
copying cp.c to temp/cp.c
copying cp.exe to temp/cp.exe

```

表 18.3 POSIX 目录访问函数

函 数	描 述
chdir	改变当前的工作目录
closedir	停止读目录
getcwd	获取当前工作目录的名字
mkdir	创建一个新目录
opendir	用读的方式打开目录
readdir	获取下一个目录的入口
rewinddir	把用于读目录的指针重新设置到文件开头
rmdir	删除一个目录（这个目录必须为空）

程序清单 18.11 列出当前目录的条目

```

/* list.c: 打印目录清单 */
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <string.h>
#include <assert.h>

static char *attr_str(short attr);

main()
{
    DIR *dirp = opendir("."); /* 当前目录 */
    struct dirent *entry;

    assert(dirp);
    while ((entry = readdir(dirp)) != NULL)
    {
        struct stat finfo;

        stat(entry->d_name, &finfo);
    }
}

```

```

        printf(
            "%-12.12s  %s %8ld %s",
            strlwr(entry->d_name),
            attr_str(finfo.st_mode),
            finfo.st_size,
            ctime(&finfo.st_mtime)
        );
    }
    closedir(dirp);
    return 0;
}

static char *attr_str(short attr)
{
    static char s[4];

    strcpy(s, "--");
    if (attr & S_IFDIR)
        s[0] = 'd';
    if (attr & S_IREAD)
        s[1] = 'r';
    if (attr & S_IWRITE)
        s[2] = 'w';
    return s;
}

/* 示例输出:*/
.          dr-      0  Fri May 02 04:13:06 1997
..         dr-      0  Fri May 02 04:05:26 1997
cat.c      -rw      875  Fri May 02 03:50:26 1997
cat.exe    -rw     61440  Fri May 02 03:50:31 1997
cp.c       -rw      1072  Fri May 02 04:00:51 1997
cp.exe     -rw     61440  Fri May 02 04:02:02 1997
filecopy.c -rw       970  Fri May 02 04:01:54 1997
list.c     -rw       968  Fri May 02 04:12:04 1997
list.exe   -rw     61440  Fri May 02 04:12:21 1997

```

前两个条目是特殊的：“.”指的是当前的目录，“..”指的是它的父目录。这些都由文件系统创建，因此不能直接地改变他们（因此没有显示‘w’访问许可）。在 POSIX 函数中根目录由标记/来表示（在 MSDOS 中也可选择用\来表示）。由于修改时间是标准的时间值（time_t），因此我用标准 C 函数 ctime 来显示时间（第 19 章解释了标准 C 的时间函数。）

18.10 重定向标准错误

当输入如下这种命令：

```
cat file1 file2 >file3
```

命令外壳把标准输出的内部文件句柄从控制台断开，并且在这个句柄装载程序 `cat` 之前把它和 `file3` 相连。当 `cat` 结束后外壳程序把这个句柄重新和控制台相连。程序清单 18.12 说明了如何用 `stderr` 来完成这项工作。函数 `vreaddir_stderr` 做了如下事情：

1. 通过调用 `open` 来获得新目标文件的句柄；
2. 用 `dup` 来创建原始目标文件的新句柄(它用于以后的恢复)；
3. 通过把句柄和原始目标文件断开并且和新目标文件相连来重定向输出（这就是 `dup2` 所做的事情）。

当不再需要任何重定向的时候，调用 `restore_stderr`：

1. 把标准错误重定向回它的原始目标文件；
2. 舍弃所拷贝的句柄。

用 `dup` 创建的句柄和原来的句柄“同步”，因此如果对一个句柄用 `lseek` 改变文件定位的话，另一个句柄的定位也得到了更新。

你可能觉得奇怪，为什么不能只通过调用 `freopen` 来重定向标准错误呢。在单独一个程序内这样做是有效的，但是如果在程序内初始化另一个程序（使用 `system`）`freopen` 就不起作用了，因为只有局部文件指针改变了。为了使这种改变在整个子进程中持续必须利用 `dup2`。

程序清单 18.12 重新定向标准错误

```
/* stderr.c: 重新定向标准错误 */
#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

static int old_handle = -1;

int redir_stderr(char *fname)
{
    int fd = open(fname, O_WRONLY|O_CREAT|O_TEXT, S_IWRITE);
    if (fd >= 0)
    {
        int err_handle = fileno(stderr);
        old_handle = dup(err_handle);
        dup2(fd, err_handle);
        close(fd);
    }
    return fd;
}

void restore_stderr(void)
{

```



```

    if (old_handle != -1)
    {
        dup2(old_handle,fileno(stderr));
        close(old_handle);
        old_handle = -1;
    }
}

```

在程序清单 18.13 中的程序 `ddir.c` 说明了前面所讨论的大部分 POSIX 概念。这个程序通过下面的步骤来删除一个完整的目录树：

1. 使树的根目录为当前的工作目录；
2. 用外壳命令删除这个目录中的所有文件（在这个程序中用 MSDOS 的 `del` 命令）；
3. 在目录中留下的条目是下面之一：
 - a. 受保护的文件——用 `chmod` 来降低保护程度并用 `unlink` 显式地删除文件（在 MSDOS 中可以用 `remove`，但是不能在 UNIX 中用这个命令。`unlink` 在两个系统中都可以使用）；
 - b. 目录——对子目录递归地重复从步骤 1 开始的整个过程。
4. 追溯到父目录并且用 `rmdir` 删除正被讨论的目录。

程序清单 18.13 删除一个目录树

```

/* ddir.c: 删除子目录树(只能在 Borland C 编译器上使用) */
#include <stdio.h>
#include <stdlib.h>
#include <io.h>
#include <sys/stat.h>
#include <dirent.h>
#include <dir.h>

main(int argc, char **argv)
{
    void rd(char*);
    char *old_path = getcwd(NULL,64);

    /* 删除目录 */
    while (argc)
        rd(++argv);

    /* 恢复目录 */
    chdir(old_path);
    free(old_path);
    return 0;
}

void rd(char* dir)
{

```

```
extern int redir_stderr(char* fname);
extern void restore_stderr(void);
void erase_dir(void);

/* Log onto the directory to be deleted */
//进入到将被删掉的目录中
chdir(dir);
/* 通过 OS 外壳 (DOS 版) 删除全部的正常文件 */
redir_stderr("nul");
system("del /q *.* > nul");
restore_stderr();

/* 删除所有剩下来的目录入口 */
erase_dir();

/* 从其父目录中删除目录 */
chdir("..");
rmdir(dir);
}

static void erase_dir(void)
{
    DIR* dirp;
    struct dirent* entry;
    struct stat finfo;

    /* 删去在当前目录中剩下的部分*/
    dirp = opendir(".");
    while ((entry = readdir(dirp)) != NULL)
    {
        if (entry->d_name[0] == '.')
            continue;
        stat(entry->d_name, &finfo);
        if (finfo.st_mode & S_IFDIR)
            rd(entry->d_name);      /* Subdirectory */
        else
        {
            /* Enable delete of file, then do it */
            chmod(entry->d_name, S_IWRITE);
            unlink(entry->d_name);
        }
    }
    closedir(dirp);
}
```

如果任何文件不能被删除 (或如果没有要删除的文件), MSDOS 的某些版本向标准错误

发送消息，我在调用系统之前将标准错误重新定向到 nul，因此这些消息不会出现。

18.11 封装重定向操作

通常，C++可以使生活变得更简单。程序清单 18.14 和程序清单 18.15 是一个 Directory 类的定义，它隐藏了 POSIX 数据结构的细节。构造函数自动打开目录并在第一个条目处进行定位，而析构函数关闭目录。程序清单 18.16 和程序清单 18.17 显示了如何在程序 list 和程序 ddir 中直接使用这个类。

程序清单 18.14 Directory 类的头文件

```
// mydir.h: 目录导航类
#include <string>
#include <exception>
#include <sys/stat.h>
#include <dirent.h>
#include <dir.h>
#include <assert.h>

using std::string;
using std::exception;

class DirError : public exception
{
public:
    DirError(const string& s) : exception(s)
    {}
};

class Directory
{
public:
    // 构造函数和析构函数:
    Directory(const string&);
    ~Directory();

    // 导航器:
    void rewind();
    void next();
    bool eof() const;

    // 目录入口的信息:
    string entry_name() const;
    long entry_size() const;
    int entry_attrs() const;
```

```
    time_t entry_time() const;
    bool entry_isdir() const;

private:
    DIR* dirp;
    struct dirent* entry;
    struct stat info;

    void do_stats();
};

inline Directory::~Directory()
{
    assert(dirp);
    closedir(dirp);
}

inline bool Directory::eof() const
{
    return entry == NULL;
}

inline void Directory::next()
{
    assert(dirp);
    entry = readdir(dirp);
    if (entry)
        do_stats();
}

inline void Directory::rewind()
{
    assert(dirp);
    rewinddir(dirp);
    next();
}

inline bool Directory::entry_isdir() const
{
    assert(dirp);
    return info.st_mode & S_IFDIR;
}
```

程序清单 18.15 Directory 类的实现文件

// mydir.cpp

```

#include "mydir.h"
Directory::Directory(const string& s)
{
    dirp = opendir(s.c_str());
    if (!dirp)
        throw DirError("invalid directory");
    next();
}
string Directory::entry_name() const
{
    assert(dirp);
    if (eof())
        throw DirError("no entry available");
    return entry->d_name;
}
int Directory::entry_attrs() const
{
    assert(dirp);
    if (eof())
        throw DirError("no entry available");
    return info.st_mode;
}
long Directory::entry_size() const
{
    assert(dirp);
    if (eof())
        throw DirError("no entry available");
    return info.st_size;
}
time_t Directory::entry_time() const
{
    assert(dirp);
    if (eof())
        throw DirError("no entry available");
    return info.st_mtime;
}
void Directory::do_stats()
{
    if (stat(entry->d_name, &info) == -1)
        throw DirError("file info not available");
}

```

程序清单 18.16 程序清单 18.10 的 C++ 版本

```

// list.cpp: 打印目录清单
#include <iostream>

```

```
#include <iomanip>
#include <sys/stat.h>
#include <time.h>
#include <string>
#include "mydir.h"

using namespace std;

string attr_str(int attr);

main()
{
    Directory d(".");
    while (!d.eof())
    {
        time_t t = d.entry_time();
        cout.setf(ios::left, ios::adjustfield);
        cout << setw(15) << d.entry_name().substr(0,12);
        cout << setw(4) << attr_str(d.entry_attrs());
        cout.setf(ios::right, ios::adjustfield);
        cout << setw(8) << d.entry_size() << " ";
        cout << ctime(&t);    //包括换行
        d.next();
    }
}

string attr_str(int attr)
{
    string result(attr & S_IFDIR ? "d" : "-");
    result += attr & S_IREAD ? "r" : "-";
    result += attr & S_IWRITE ? "w" : "-";
    return result;
}
```

程序清单 18.17 程序清单 18.13 的 C++ 版本

```
// ddir.cpp
#include <iostream>
#include <string>
#include <stdexcept>
#include <stdlib.h>
#include <dir.h>
#include <io.h>
#include "mydir.h"

using namespace std;
```

```
extern "C" int redir_stderr(char *);
extern "C" void restore_stderr(void);

main(int argc, char **argv)
{
    char *old_path = getcwd(NULL, 64);
    void rd(const string&);

    try
    {
        while (argc)
            rd(*++argv);
    }
    catch (exception& x)
    {
        cerr << x.what() << endl;
        return EXIT_FAILURE;
    }

    chdir(old_path);
    free(old_path);
    return EXIT_SUCCESS;
}

void rd(const string& dir)
{
    void erase_dir(void);

    chdir(dir.c_str());
    redir_stderr("nul");
    system("del /q *.* > nul");
    restore_stderr();

    erase_dir();

    chdir("..");
    rmdir(dir.c_str());
}

static void erase_dir(void)
{
    Directory d(".");
    for (; !d.eof(); d.next())
    {
        string dname = d.entry_name();
```

```
    if (dname[0] == '.')
        continue;
    if (d.entry_isdir())
        rd(dname);
    else
    {
        chmod(dname.c_str(), S_IWRITE);
        unlink(dname.c_str());
    }
}
```

18.12 小结

- 不是所有的文件操作端口都能跨平台。那些通常不受限制的操作都在标准 C 和 C++ 库中。
- 过滤器只能从标准输入中读入并只能向标准输出写。过滤器在支持 I/O 重定向的环境下是非常强大的工具。
 - 当写文件的时候不要忘记检查输出——你可能使输出介质泛滥。
 - 某些环境区分文本文件和二进制文件。块 I/O 和随机文件定位需要二进制模式。
 - 如果标准库不能满足文件处理的需要，试着用 POSIX 文件操作。如果还不行，把依赖平台的代码分隔在独立的模块中以利于端口进程。
- 总是考虑用 C++ 来封装低级别的概念和操作。

时间和日期处理

大多数操作系统都有对当前日期和时间进行跟踪的方法。C 语言通过在<time.h>中定义库函数可以得到这种信息的各种格式。time 函数返回一个 time_t 类型的值（通常是长整型 long），它是由实现决定的对当前日期和时间的编码。为了得到更多明确的信息，可以把这个值传递给其他函数如 localtime，下面的例子以各种格式打印当前的日期和时间。

```
/* tformat.c */
#include <stdio.h>
#include <time.h>

main()
{
    time_t tval;
    struct tm *now;
    char buf[BUFSIZ];
    char *fancy_format =
        "Or getting really fancy:\n"
        "Today is %A, %B %d, day %j of %Y,\n"
        "and the time is %I:%M %p";

    /* 获得当前的日期和时间*/
    tval = time(NULL);
    now = localtime(&tval);

    printf("The current date and time: %d/%02d/%02d %d:%02d:%02d\n\n",
        now->tm_mon+1, now->tm_mday, now->tm_year,
        now->tm_hour, now->tm_min, now->tm_sec);
    printf("Or in default system format: %s\n", ctime(&tval));
    strftime(buf, sizeof buf, fancy_format, now);
```

```
    puts(buf);
    return 0;
}

/*输出: */
The current date and time: 3/27/97 16:58:11
Or in default system format: Thu Mar 27 16:58:11 1997

Or getting really fancy:
Today is Thursday, March 27, day 086 of 1997,
and the time is 04:58 PM
```

localtime 函数把编码的时间分解成下面的成分:

```
struct tm
{
    int  tm_sec;        /*秒 (0-60) */
    int  tm_min;        /*分 (0-59) */
    int  tm_hour;       /*一天的第几个小时 (0-23) */
    int  tm_mday;       /*一个月的第几天 (1-31) */
    int  tm_mon;        /*从1月份开始的月份!!! */
    int  tm_year;       /*从1990年开始的年*/
    int  tm_wday;       /*一个星期的第几天 (0-6) */
    int  tm_yday;       /*从1月1号开始的一年中的第几天 (0-365) */
    int  tm_isdst;      /*白天的保存是否有效*/
}
```

localtime 函数返回一个指向静态结构的指针, 因此在没有复制时程序中每次只有一个这样的结构可以使用。**ctime** 函数返回一个指向静态字符串的指针, 该字符串以标准的格式描述完整的时间和日期并以换行符结束。**strftime** 按照用户规范格式化一个字符串。例如, 描述符 **%A** 表示在一星期中某天的名称。(格式描述符的完整列表见表 19.1)。

下面的示例程序显示了如何编写简单的日期/时间算法:

```
/* tmath.c: 计算未来日期和已消耗的程序时间 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

main()
{
    time_t start, stop;
    struct tm *now;
    int ndays;

    /* 获得当前的日期和时间 */
    time(&start);
    now = localtime(&start);
```

```

/*在天数间插入一个时间间隔*/
fputs("How many days from now? ",stdout);
fflush(stdout);
if (scanf("%d", &ndays) != 1)
    return EXIT_FAILURE;
now->tm_mday += ndays;
if (mktime(now) != -1)
    printf("\nNew date: %s", asctime(now));
else
    puts("Sorry. Can't encode your date.");

/* 计算程序所用时间*/
time(&stop);
printf("Elapsed program time in seconds: %f\n",
        difftime(stop,start));
return EXIT_SUCCESS;
}
/* 输出: */
How many days from now? 45
New date: Sun May 11 17:01:47 1997
Elapsed program time in seconds: 1.000000

```

同时也要注意上面例子中 `time` 函数交替的语义(即 `time_t` 类型参数的地址被传递而不是作为返回值使用)。`mktime` 函数改变一个 `tm` 结构以使日期值和时间值都是在适当的范围内,之后它更新了星期的天数域(`tm_wday`)和年的天数域(`tm_yday`)。如果所指定的日期不能被表示,则 `mktime` 函数将会失败。当正被讨论的日期先于实现的参考日期时,会发生这种情况。例如,一个典型的基于 DOS/Windows 的编译器不能对 1970 年 1 月 1 日前的日期进行编码,但 VAX C 的参考日期却是在 19 世纪中期。远在未来的日期也会引起编码失败。在编译器参考手册中的 `time` 描述可以查到所使用的平台的限制。`asctime` 函数返回在传递的 `tm` 结构中代表时间的标准字符串,所以 `ctime(&tval)` 等于 `asctime(localtime(&tval))`。`difftime` 函数以 `double` 型返回两个编码中相差的秒数。

表 19.1

strftime 格式描述符

<code>%A, %a</code>	完整的/缩写的星期名 (Saturday/Sat)
<code>%B, %b</code>	完整的/缩写的月份名 (April/Apr)
<code>%c</code>	日期和时间 (Apr 12 12:38:03 1997)
<code>%d</code>	一个月的第几天 (12)
<code>%H</code>	小时 (24 小时制: 12)
<code>%I</code>	小时 (12 小时制: 12)
<code>%j</code>	一年中的第几天 (102)
<code>%m</code>	月份 (从一月开始: 4)

%M	1 小时的第几分钟 (38)
%p	AM/PM (PM)
%s	1 分钟的第几秒 (03)
%U	一年的第几个星期(以星期日为一个星期开始: 14)
%w	星期几 (星期日是 0: 6)
%W	一年的第几个星期(以星期一为一个星期开始: 14)
%x	日期 (Apr 12 1997)
%X	时间 (12::38:03)
%y	一个世纪的第几年 (97)
%Y	年 (1997)
%Z	时区 (如果支持的话)

有时可以在系统参考范围之外处理日期, 或用秒以外的单位来计算两个日期之间的间隔。在这种情况下, 你应该设计自己的日期编码。在程序清单 19.1~19.3 中的程序显示了一种使用简单的年/月/日编码来确定两个日期之间的年数、月及月数和日数的技术(注意考虑了闰年)。为了简短起见, 该程序假设日期有效并且第一个日期在第二个日期之前。随着 `time.h` 中函数的引导, `date_interval` 函数返回一个指向静态 `Date` 结构体的指针。

程序清单 19.1 一个简单的日期结构

```
/* date.h: 日期的简单编码*/
struct DATE
{
    int year;      /* 全部日期(例如,1992, 而不是 92) */

    int month;     /* 1 - 12 */
    int day;       /* 1 - 31 */
};
typedef struct DATE Date;

Date* date_interval(Date *, Date *);
```

程序清单 19.2 计算两日期之间的持续时间

```
/* date.c: 计算两日期之间的持续时间 */
#include <assert.h>
#include "date.h"

#define isleap(y) ((y)%4 == 0 && (y)%100 != 0 || (y)%400 == 0)

/* 平年和闰年每月的天数: */
static int Daytab[2][13] =
```

```

{
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
}; continued
Date* date_interval(Date *d1, Date *d2)
{
    static Date result;
    int months, days, years, prev_month, year2;

    /* 计算间隔——假定 d1 领先 d2 */
    years = d2->year - d1->year;
    assert(years > 0);
    months = d2->month - d1->month;
    days = d2->day - d1->day;

    /* 做明显的修正(在月前必须调整天数!)
     * 倘若前一个月是二月则循环,
     * 并且天数 < -28.
     */
    prev_month = d2->month - 1;
    year2 = d2->year;
    while (days < 0)
    {
        /* 从前一个月借 */
        if (prev_month == 0)
        {
            prev_month = 12;
            --year2;
        }
        --months;
        days += Daytab[isleap(year2)][prev_month--];
    }

    if (months < 0)
    {
        /* 从前一年借 */
        --years;
        months += 12;
    }
    /* 准备输出 */
    result.month = months;
    result.day = days;
    result.year = years;
    return &result;
}

```

程序清单 19.3 测试日期间隔函数

```

/* tdate.c: 测试 date_int() */
#include <stdio.h>
#include "date.h"

main()
{
    Date d1, d2, *result;

    /* 读入两个日期——假定第一日先于第二日 */
    fputs("Enter a date, MM/DD/YY> ", stdout);
    fflush(stdout);
    if (scanf("%d/%d/%d%c", &d1.month, &d1.day, &d1.year) != 3)
        return 1;

    fputs("Enter a later date, MM/DD/YY> ", stdout);
    fflush(stdout);
    if (scanf("%d/%d/%d%c", &d2.month, &d2.day, &d2.year) != 3)
        return 1;

    /* 计算年、月和日的间隔 */
    result = date_interval(&d1, &d2);
    printf("years: %d, months: %d, days: %d\n",
        result->year, result->month, result->day);
    return 0;
}

/* 输出: */
Enter a date, MM/DD/YY> 5/1/1954
Enter a later date, MM/DD/YY> 3/27/1997
years: 42, months: 10, days: 26

```

19.1 Julian 日期编码

当 `date_interval` 函数适应我们通常关于时代的概念时，日期算法中经常要求更高的精确度。例如，银行要求计算每天的利息。天文学家利用 Julian 日期编码来简化日期计算。Julian Day 算法被 Joseph Scaliger（大约公元 1577 年）发展成为一种给所记录时间的天数分配连续数字的方法。某天的 Julian 日期数是自公元前 4713 年 1 月 1 日午夜起过去的日期数。（在这里“Julian”这个名字不是来自于 Julian 日历，而是来自于 Scaliger 的父亲的名字）。

Julian 日历系统起源于公元前 45 年，我们的 Gregorian 日历就是以此为基础的。Julian 日历假设一年中有 365 天，因此定义每年有 365 天，每 4 年有一个闰年。然而，经证实回归年

每年都在变化，实际上平均一年接近 365.242 天。到 1582 年 Julian 日历系统的错误变得十分明显，春分比时间表中的时间提前了大约 10 天。就在那时决定了一个可以和自然保持同步的更好的方法，那就是修正闰年标准——忽略能被 100 整除的年，除了那些能被 400 整除的年，这就是程序清单 19.2 中的 `isleap` 公式。为了使一切回到正确的轨道上来，Gregory 教皇在公元 1582 年 10 月 4 日星期四宣布结束 Julian 日历，第二天被命名为公元 1582 年 10 月 15 日星期五，作为 Gregory 日历时代的开端。

程序清单 19.4 到 19.6 定义了大量的日期处理函数，包括 `GregToJul` 和 `JulToGreg` 函数，它们分别实现 Gregory 日历和 Julian 日历之间的互换。这里所使用的算法根据 *C User Journal* 中的文章改编，并且仅使用整数运算，因此这个算法比用实数将一天分成两部分的天文学的版本更有效。这些算法对在 1582 年 10 月 15 日之后的所有日期都是准确的。对于早于这个日期的年份，它们与真实的天文学数据相差 10 天，与公元前的日期相差 1 年。`JulToGreg` 能毫无错误地恢复任何被 `GregToJul` 改变了的原始日期。然而，对于在 1582 年 10 月 15 日之前的任何日期使用该函数的话，如用两个 Julian 日期数相减来确定过去的日期数或者调用 `DayOfWeek`，可能会产生错误结果。为了简单起见，我不使用 1583 年之前的任何日期。程序清单 19.7 说明了来自于 `DateStuff` 名字空间的各种函数。

程序清单 19.4 日期类和函数的声明

```
// datefwd.h
#ifndef DATEFWD_H
#define DATEFWD_H

#include <limits.h>
#include <string>

namespace DateStuff
{
    // 常数和数据:
    enum YearLimits {MIN_YEAR = 1583, MAX_YEAR = INT_MAX};
    enum Days {MONDAY, TUESDAY, WEDNESDAY, THURSDAY,
               FRIDAY, SATURDAY, SUNDAY, DAY_ERROR};
    enum Months {MONTH_ERROR, JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
                 JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER};

    // 类: (稍后再解释)
    struct Duration;
    class Year;
    class YMonth;
    class Date;
    class FullDate;
    class Date_Exception;
```

// 常用的日期函数:

// 系统日期:

```
void Today(int&, int&, int&);
```

// 闰年测试:

```
bool IsLeap(int y);
```

// 其他的有效性检查:

```
bool IsValidYear(int y);
```

```
bool IsValidMonth(int m);
```

```
bool IsValidDay(int m, int d);
```

```
bool IsValidYMDay(int y, int m, int d);
```

// 不同的计算方法:

```
int Compare(int y1, int m1, int d1,  
            int y2, int m2, int d2);
```

```
int DayOfWeek(int y, int m, int d);
```

```
int DayOfYear(int y, int m, int d);
```

```
int EndOfMonth(int y, int m);
```

```
int EndOfYear(int y);
```

```
void NthDay(int n, int y, int& m, int& d);
```

```
int NthWeekDay(int n, int wd, int y, int m);
```

```
int DaysInPrevMonth(int y, int m);
```

```
int DaysInNextMonth(int y, int m);
```

```
int GetDaysInMonth(bool isLeap, int m);
```

```
int GetDaysToDate(bool isLeap, int m);
```

```
int FirstSat(int dow);
```

```
int FirstSatOfMonth(int y, int m);
```

```
int FirstSatOfYear(int y);
```

```
int AbsoluteWeek(int doy);
```

```
int AbsoluteWeekOfMonth(int y, int m, int d);
```

```
int AbsoluteWeekOfYear(int y, int m, int d);
```

```
int CommonWeek(int s, int doyom);
```

```
int CommonWeekOfMonth(int y, int m, int d);
```

```
int CommonWeekOfYear(int y, int m, int d);
```

```
void NthCommonWeek(int n, int y, int& doy1, int& doy2);
```

```
int NumCommonWeeks(int y);
```

```
void AddYears(int yrs, int& y, int& m, int& d);
```

```
void SubtractYears(int yrs, int& y, int& m, int& d);
```

```
void AddMonths(int mths, int& y, int& m, int& d);
```

```
void SubtractMonths(int mths, int& y, int& m, int& d);
```



```

long MonthsBetween(int y1, int m1, int y2, int m2);
void AddDays(int days, int& y, int& m, int& d);
void SubtractDays(int days, int& y, int& m, int& d);
long DaysBetween(int y1, int m1, int d1, int y2, int m2, int d2);
void AddWeekDays(int wdays, int& y, int& m, int& d);
void SubtractWeekDays(int wdays, int& y, int& m, int& d);
long WeekDaysBetween(int y1, int m1, int d1, int y2, int m2, int d2);
void AddWeeks(int wks, int& y, int& m, int& d);
void SubtractWeeks(int wks, int& y, int& m, int& d);
long WeeksBetween(int y1, int m1, int d1, int y2, int m2, int d2);
continued
Duration Age(int y, int m, int d);
Duration AgeBetween(int y1, int m1, int d1,
                    int y2, int m2, int d2);

// Julian 日期变换:
long GregToJul(int y, int m, int d);
void JulToGreg(long jd, int& y, int& m, int& d);
int DayOfWeek(long jd);
int DayOfYear(long jd);

// 其他的变换:
using std::string;
string ToString(int y, int m, int d);
void FromString(const string& str, int& y, int& m, int& d);

//其他
void ResolveMonths(long mths, int& y, int& m);
void CheckY(int y);
void CheckYM(int y, int m);
void CheckYMD(int y, int m, int d);
}

#endif

```

程序清单 19.5 Date 功能模块

```

// DateStuff.h:
#ifndef DATESTUF_H
#define DATESTUF_H

#include <assert.h>
#include "datefwd.h"
#include "xcept.h" using std::string;

namespace DateStuff

```

```
{

    //类:
    struct Duration
    {   int m_Year;
        int m_Month;
        int m_Day;
        Duration(int y = 0, int m = 0, int d = 0)
        {
            m_Year = y;
            m_Month = m;
            m_Day = d;
        }
    };

    // 异常类:
    using std::exception;
    class Date_Exception : public exception
    {
    public:
        Date_Exception(int cod, const string& msg = "")
            : exception((s_ErrorStrings[cod] + ":" + msg).c_str())
        {
            assert(BEGIN <= cod && cod < END);
        }
        enum {BEGIN = 200};
        enum {DATE_ERROR = BEGIN, YEAR_ERROR, MONTH_ERROR,
              DAY_ERROR, RANGE_ERROR, BIRTHDAY_ERROR,
              END, NUM_ERRORS = END - BEGIN};

    protected:
        string ErrorString(int) const;    //重载

    private:
        static string s_ErrorStrings[NUM_ERRORS];
    };

    //内联:
    inline bool IsLeap(int y)
    {

        // 确定是否是闰年(返回 1 或 0)
        return !(y % 4 == 0 && y % 100 != 0 || y % 400 == 0);
    }
}
```

```
inline bool IsValidYear(int y)
{
    return MIN_YEAR <= y && y <= MAX_YEAR;
}

inline bool IsValidMonth(int m)
{
    return 1 <= m && m <= 12;
}

inline bool IsValidYMonth(int y, int m)
{
    return IsValidYear(y) && IsValidMonth(m);
}

inline bool IsValidDay(int m, int d)
{
    return IsValidMonth(m) &&
        1 <= d && d <= GetDaysInMonth(true,m);
}

inline bool IsValidYMDay(int y, int m, int d)
{
    return IsValidYMonth(y,m) &&
        1 <= d && d <= EndOfMonth(y, m);
}

inline int DayOfWeek(long jd)
{
    return int(jd % 7L); //看上面的枚举日期
}

inline int DayOfWeek(int y, int m, int d)
{
    CheckYMD(y,m,d);
    return int(GregToJul(y, m, d) % 7);
}

inline int DayOfYear(long jd)
{
    int y, m, d;
    JulToGreg(jd, y, m, d);
    return DayOfYear(y, m, d);
}

inline int DayOfYear(int y, int m, int d)
```

```
{
    CheckYMD(y,m,d);
    return int(GetDaysToDate(IsLeap(y),m) + d);
}

inline int EndOfMonth(int y, int m)
{
    CheckYM(y,m);
    return GetDaysInMonth(IsLeap(y),m);
}

inline int EndOfYear(int y)
{
    CheckY(y);
    return GetDaysToDate(IsLeap(y),13);
}

inline int AbsoluteWeek(int d)
{
    assert(1 <= d && d <= 366);
    return (d - 1) / 7 + 1;
}

inline int AbsoluteWeekOfYear(int y, int m, int d)
{
    CheckYMD(y,m,d);
    return AbsoluteWeek(DayOfYear(y,m,d));
}

inline int AbsoluteWeekOfMonth(int y, int m, int d)
{
    CheckYMD(y,m,d);
    return AbsoluteWeek(d);
}

inline int CommonWeekOfMonth(int y, int m, int d)
{
    CheckYMD(y,m,d);
    return CommonWeek(FirstSatOfMonth(y,m), d);
}

inline int CommonWeekOfYear(int y, int m, int d)
{
    CheckYMD(y,m,d);
    return CommonWeek(FirstSatOfYear(y), DayOfYear(y,m,d));
}
```

```

inline int FirstSatOfYear(int y)
{
    CheckY(y);
    return FirstSat(DayOfWeek(y,1,1));
}

inline int FirstSatOfMonth(int y, int m)
{
    CheckYM(y,m);
    return FirstSat(DayOfWeek(y,m,1));
}

inline int Compare(int y1, int m1, int d1, int y2, int m2, int d2)
{
    // 返回两日期中间天数 (有符号数)
    return GregToJul(y1,m1,d1) - GregToJul(y2,m2,d2);
}

inline long DaysBetween(int y1, int m1, int d1, int y2, int m2, int d2)
{
    // 用于比较的同义词
    return Compare(y1,m1,d1,y2,m2,d2);
} continued
inline int NumCommonWeeks(int y)
{
    CheckY(y);
    return (EndOfYear(y) - FirstSatOfYear(y) - 1) / 7 + 2;
}

inline long MonthsBetween(int y1, int m1, int y2, int m2)
{
    CheckYM(y1,m1);
    CheckYM(y2,m2);
    return (y1 - y2)*12L + m1 - m2;
}

inline void AddDays(int days, int& y, int& m, int& d)
{
    CheckYMD(y,m,d);
    JulToGreg(GregToJul(y,m,d) + days, y, m, d);
}
inline void SubtractDays(int days, int& y, int& m, int& d)
{
    CheckYMD(y,m,d);
    JulToGreg(GregToJul(y,m,d) - days, y, m, d);
}

```

```
    }

    inline void AddWeeks(int wks, int& y, int& m, int& d)
    {
        CheckYMD(y,m,d);
        AddDays(wks*7,y,m,d);
    }

    inline void SubtractWeeks(int wks, int& y, int& m, int& d)
    {
        CheckYMD(y,m,d);
        SubtractDays(wks*7,y,m,d);
    }

    inline long WeeksBetween(int y1, int m1, int d1, int y2, int m2, int d2)
    {
        CheckYMD(y1,m1,d1);
        CheckYMD(y2,m2,d2);
        return DaysBetween(y1,m1,d1,y2,m2,d2) / 7L;
    }

}          // 结束命名空间 DateStuff
#endif
```

程序清单 19.6 Date 功能模块(续)

```
// datestuff.cpp
#include "DateStuff.h"
#include <time.h>
#include <stdlib.h>
#include <algorithm>    // 为 min(), swap()

using std::min;

namespace DateStuff
{
    static const int DaysInMonth[][13] =
    {
        {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
        {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
    };

    static const int DaysToDate[][13] =
    {
        {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365},
        {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366}
    }
}
```

```

    };
}

int DateStuff::GetDaysInMonth(bool isLeap, int m)
{
    assert(isLeap == 0 || isLeap == 1);
    if (!IsValidMonth(m))
        Throw(Date, MONTH_ERROR);
    return DaysInMonth[isLeap][m];
}

int DateStuff::GetDaysToDate(bool isLeap, int m)
{
    //m 必须在 [1,13] 间
    assert(isLeap == 0 || isLeap == 1);
    if (m < 1 || 13 < m)
        Throw(Date, MONTH_ERROR);
    return DaysToDate[isLeap][m-1];
}

long DateStuff::GregToJul(int year, int month, int day)
{
    // 转换阳历日期 (Gregorian) 为完整的 Julian 成员
    CheckYMD(year, month, day);
    long m = long(month);
    long d = long(day);
    long y = long(year);
    continued
    return d - 32075L
        + 1461L * (y + 4800 + (m - 14L)/12L) / 4L
        + 367L * (m - 2L - (m - 14L)/12L * 12L) / 12L
        - 3L * ((y + 4900L + (m - 14L)/12L) / 100L) / 4L;
}

void DateStuff::JulToGreg(long jday, int& year, int& month, int& day)
{
    // 转换 Julian 天数为阳历日期 (Gregorian)
    long t1 = jday + 68569L;
    long t2 = 4L*t1 / 146097L;
    t1 -= (146097L*t2 + 3L) / 4L;

    long y = 4000L*(t1 + 1)/1461001L;
    t1 = t1 - 1461L*y/4L + 31;

    long m = 80L *t1/2447L;

```

```
    day = int(t1 - 2447L*m/80L);

    t1 = m / 11L;
    month = int(m + 2L - 12L*t1);

    year = int(100L * (t2 - 49L) + y + t1);
}

void DateStuff::NthDay(int n, int year, int& month, int& day)
{
    CheckY(year);
    if (n < 1 || 366 < n)
        Throw(Date, DAY_ERROR);

    int row = IsLeap(year);
    if (n > DaysToDate[row][12])
        Throw(Date, RANGE_ERROR);

    for (month = 0; month < 13; ++month)
        if (DaysToDate[row][month] >= n)
            break;

    if (DaysToDate[row][month] > n)
        day = int(n - DaysToDate[row][month-1]);
    else
        day = int(DaysToDate[row][month] - DaysToDate[row][month-1]);
}

int DateStuff::NthWeekDay(int n, int weekDay, int year, int month)
{
    CheckYM(year, month);
    if (!(1 <= n && n <= 5) ||
        !(MONDAY <= weekDay && weekDay <= SUNDAY))
        Throw(Date, DAY_ERROR);

    // 查找所需的星期中一天第一次出现的地方:
    long jday = GregToJul(year, month, 1);
    while (int(jday % 7L) != weekDay)
        ++jday;

    // 被所需的星期中的一天第 n 个出现的超前星期数:
    for (int weekno = 1; weekno < n; ++weekno)
        jday += 7L;

    // 转换 Julian 为阳历并将该天返回
    int day;
```



```

        int tempmonth = month;
        JulToGreg(jday, year, month, day);
        if (month != tempmonth)
            Throw(Date, RANGE_ERROR);
        return day;
    }

    int DateStuff::DaysInPrevMonth(int year, int month)
    {
        CheckYM(year, month);
        if (month == 1)
        {
            --year;
            month = 12;
        }
        else
            --month;
        return DaysInMonth[IsLeap(year)][month];
    }

    int DateStuff::DaysInNextMonth(int year, int month)
    {
        CheckYM(year, month);
        if (month == 12)
        {
            ++year;
            month = 1;
        }
        else
            ++month;
        return DaysInMonth[IsLeap(year)][month];
    }

    string
    DateStuff::Date_Exception::s_ErrorStrings[Date_Exception::NUM_ERRORS] =
    {
        "Invalid Date",
        "Invalid Year",
        "Invalid Month",
        "Invalid Day",
        "Range error",
        "Invalid Birthday"
    };

    string DateStuff::Date_Exception::ErrorString(int cod) const
    {

```

```

    assert(BEGIN <= cod && cod < END);
    return s_ErrorStrings[cod-BEGIN];
}

int DateStuff::FirstSat(int dow)
{
    // 计算第一个星期六的序数,
    // 已知 (一个月或一年的) 第一天是星期几
    // 注意: 假定 MONDAY == 0 ... SUNDAY == 6
    assert(MONDAY <= dow && dow <= SUNDAY);
    if (dow == SUNDAY)
        return 7;
    else if (dow == SATURDAY)
        return 1;
    else
        return SATURDAY - dow + 1;
}

int DateStuff::CommonWeek(int s, int d)
{
    // s 是 (每月或每年的) 第一个星期六的序数。
    // d 是 (每月或每年) 相应天的序数。
    // 见: CommonWeekOfMonth, CommonWeekOfYear.
    assert(1 <= s && s <= 7);
    assert(1 <= d && d <= 366);
    if (d <= s)
        return 1;
    else
        return 2 + (d - s - 1) / 7;
}

void DateStuff::Today(int& y, int& m, int& d)
{
    time_t tval = time(0);
    struct tm *tmp = localtime(&tval);

    d = tmp->tm_mday;
    m = tmp->tm_mon + 1;
    y = tmp->tm_year + 1900;
}

void DateStuff::NthCommonWeek(int n, int y, int& doy1, int& doy2)
{
    if (!(1 <= doy1 && doy1 <= EndOfYear(y)) ||
        !(1 <= doy2 && doy2 <= EndOfYear(y)))
        Throw(Date, DAY_ERROR);
}

```

```

    int s = FirstSatOfYear(y);
    if (n <= 1)
    {
        doy1 = 1;
        doy2 = s;
    }
    else
    {
        int nc = NumCommonWeeks(y);
        if (n > nc)
            n = nc;
        doy1 = s + 1 + 7*(n - 2);
        doy2 = (n == nc) ? EndOfYear(y) : s + 7*(n-1);
    }
}

void DateStuff::AddYears(int yrs, int& y, int& m, int& d)
{
    CheckYMD(y,m,d);
    if (y > MAX_YEAR - yrs)
        Throw(Date,RANGE_ERROR);
    y += yrs;
    d = min(d, EndOfMonth(y,m));    //防止跳变
}

void DateStuff::SubtractYears(int yrs, int& y, int& m, int& d)
{
    CheckYMD(y,m,d);
    if (y < MIN_YEAR + yrs)
        Throw(Date,RANGE_ERROR);
    y -= yrs;
    d = min(d, EndOfMonth(y,m));    //防止跳变
}

void DateStuff::AddMonths(int mths, int& y, int& m, int& d)
{
    CheckYMD(y,m,d);
    ResolveMonths(y*12 + m + mths, y, m);
    d = min(d, EndOfMonth(y,m));
}

void DateStuff::SubtractMonths(int mths, int& y, int& m, int& d)
{
    CheckYMD(y,m,d);
    ResolveMonths(y*12 + m - mths, y, m);
    d = min(d, EndOfMonth(y,m));
}

```

```
}

void DateStuff::AddWeekDays(int wdays, int& y, int& m, int& d)
{
    CheckYMD(y,m,d);
    // 注意: 这种算法使用在 MONDAY == 0 的情况
    // 备份到最近的周日:
    int dayno = DayOfWeek(y,m,d);
    if (dayno > FRIDAY)
    {
        SubtractDays(dayno - FRIDAY,y,m,d);
        dayno = FRIDAY;
    }

    // 首先被星期提前:
    if (wdays >= 5)
        AddWeeks(wdays / 5,y,m,d);

    // 现在提前其余的天:
    int extra = int(wdays % 5);
    if (dayno + extra > FRIDAY)
        extra += 2;    // 跳过周末
    if (extra > 0)
        AddDays(extra,y,m,d);
}

void DateStuff::SubtractWeekDays(int wdays, int& y, int& m, int& d)
{
    CheckYMD(y,m,d);
    // 注意: 这算法使用到 MONDAY == 0 的情况
    // 提前到最近的工作日:
    int dayno = DayOfWeek(y,m,d);
    if (dayno > FRIDAY)
    {
        AddDays(7 - dayno,y,m,d);
        dayno = MONDAY;
    }

    // 首先减去星期:
    SubtractWeeks(wdays / 5,y,m,d);

    // 现在备份其余的天:
    int extra = int(wdays % 5);
    if (dayno - extra < MONDAY)
        extra += 2;
    // 跳过周末
```

```

        SubtractDays(extra,y,m,d);
    }

    long DateStuff::WeekDaysBetween(int y1, int m1, int d1,
                                     int y2, int m2, int d2)
    {
        CheckYMD(y1,m1,d1);
        CheckYMD(y2,m2,d2);

        // 备份到最近的工作日:
        int dayno1 = DayOfWeek(y1,m1,d1);
        if (dayno1 > FRIDAY)
        {
            SubtractDays(dayno1 - FRIDAY,y1,m1,d1);
            dayno1 = FRIDAY;
        }
        int dayno2 = DayOfWeek(y2,m2,d2);
        if (dayno2 > FRIDAY)
        {
            SubtractDays(dayno2 - FRIDAY,y2,m2,d2);
            dayno2 = FRIDAY;
        }

        long weeks = WeeksBetween(y1,m1,d1,y2,m2,d2);
        int extra = int(dayno2 - dayno1);
        long days = weeks*5 + extra;
        return days;
    }

    DateStuff::Duration DateStuff::Age(int y, int m, int d)
    {
        int y2, m2, d2;
        Today(y2,m2,d2);
        return AgeBetween(y,m,d,y2,m2,d2);
    }

    DateStuff::Duration DateStuff::AgeBetween(int y1, int m1, int d1,
                                              int y2, int m2, int d2)
    {
        // 查找两个日期的顺序:
        int order = Compare(y1,m1,d1,y2,m2,d2);
        if (order == 0)
            return Duration(0,0,0);
        else if (order > 0)
        {
            // 使 date1 先于 date2:

```

```
        using std::swap;
        swap(y1, y2);
        swap(m1, m2);
        swap(d1, d2);
    }

    int years = y2 - y1;
    int months = m2 - m1;
    int days = d2 - d1;
    assert(years > 0 || years == 0 && months > 0 ||
           years == 0 && months == 0 && days > 0);

    int lastMonth = m2;
    int lastYear = y2;
    while (days < 0)
    {
        // 从月借:
        assert(months > 0);
        days += DaysInPrevMonth(lastYear, lastMonth--);
        --months;

        // 这是循环防止从 2 月借 1,
        // 那样就无法获得足够的天数
        // 来使 'days' 为非负。 这个循环
        // 迭代绝不超过两次
    }

    if (months < 0)
    {
        // 向年借:
        assert(years > 0);
        months += 12;
        --years;
    }

    return Duration(years, months, days);
}

void DateStuff::ResolveMonths(long months, int& y, int& m)
{
    assert(months > 1582*12);

    if (months < 0)
        months = -months;
    y = int(months / 12);
    m = int(months % 12);
}
```

```

        if (m == 0)
            m = 12;
        if (!IsValidYMonth(y,m))
            Throw(Date,RANGE_ERROR);
    }

void DateStuff::FromString(const string& s, int& y, int& m, int& d)
{
    y = atoi(s.substr(0,4).c_str());
    m = atoi(s.substr(4,2).c_str());
    d = atoi(s.substr(6,2).c_str());
    if (!IsValidYMDay(y,m,d))
        Throw(Date,DATE_ERROR);
}

void DateStuff::CheckYMD(int y, int m, int d)
{
    if (!IsValidYMDay(y,m,d))
        Throw(Date,DATE_ERROR);
}

void DateStuff::CheckYM(int y, int m)
{
    if (!IsValidYMonth(y,m))
        Throw(Date,DATE_ERROR);
}

void DateStuff::CheckY(int y)
{
    if (!IsValidYear(y))
        Throw(Date,DATE_ERROR);
}

string DateStuff::ToString(int y, int m, int d)
{
    char buf[9];
    sprintf(buf,"%04d%02d%02d",y,m,d);
    return string(buf);
}

```

程序清单 19.7 说明各种日期函数

```
// tdstuf.cpp
```

```
#include "datestuf.h"
#include <iostream>
```

```

const char* DayText[] = {"Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday", "Sunday"};

using namespace DateStuff;

main()
{
    int year = 1997;
    int month = 4;
    int day = 1;

    cout.setf(ios::boolalpha);
    cout << "Date: " << month << '/' << day << '/' << year << endl;
    cout << "Leap year? " << IsLeapYear(year) << endl;
    cout << "Day of week: " << DayText[DayOfWeek(year, month, day)] << endl;
    cout << "Day of year: " << DayOfYear(year, month, day) << endl;
    cout << "Days in month: " << EndOfMonth(year, month) << endl;
    cout << "Days in previous month: " << DaysInPrevMonth(year, month)
        << endl;
    cout << "Days in next month: " << DaysInNextMonth(year, month) << endl;
    cout << "3rd Saturday of month: " << NthWeekDay(3, SATURDAY, year, month)
        << endl;
    long jday = GregToJul(year, month, day);
    cout << "Julian day number: " << jday << endl;
    int year2, month2, day2;
    JulToGreg(jday + 100, year2, month2, day2);
    cout << "100 days from now: " << month2 << '/' << day2 << '/' << year2
        << endl;
}

// 输出:
Date: 4/1/1997
Leap year? false
Day of week: Tuesday
Day of year: 91
Days in month: 30
Days in previous month: 31
Days in next month: 31
3rd Saturday of month: 19
Julian day number: 2450540
100 days from now: 7/10/1997

```

19.2 用于实际工作的日期类

如果定义一个 C++ 日期类, 该类包含 Julian 日期数字、或者包含年月日的组合, 有能力

决定年的哪天或哪周或是根据年月日来找到两个日期之间的持续时间，从这点上来讲可能是件容易的事。然而，商业数据处理通常有更复杂的要求。有时可能只有年或是年和月的数据，但是仍然可以在这样部分的日期中实现许多重要的功能。本章余下的部分实现了适应复杂商业要求的类，包括部分日期的处理。

在下面所描述的 `Date` 类在 32 位环境中处理了从 1583 年 1 月 1 日到 2147, 483, 647 年 12 月 31 日的 `Gregory` 日期，并提供了如下的功能：

- 检验一年是否是“闰年”
- 日期是星期几
- 日期是一年的哪一天
- 年或月中的绝对星期
- 年或月中的普通星期
- 找到年中的第 n 个普通的星期
- 一个月中有多少天
- 一年中有多少天
- 日期是在所给年的某个月中是第几个星期几（例如第三个星期三）
- 在年/月/日和数值字符串之间进行转换
- 比较相关的日期
- 向/从一个日期中加/减下列内容：
 - 年
 - 月
 - 周
 - 日
 - 星期几（从星期一到星期五）

● 计算两个日期之间的持续时间。用年代术语（年、月、日）给出答案或是如同下面的某一个：年、月、周日或星期几（例如，`YearsBetween`、`WeekDaysBetween` 等等）。

为了处理部分日期，`Date` 必须接受年、年和月，或是年月日，所以它需要一个如下的构造函数：

```
Date ( int y=0 ,int m=0, int d=0);
```

这个特殊的构造函数也允许一个空的日期，意思是没有可用的信息，它经常在实际的应用程序中发生。不管日期是否是空的、部分的或是完全的，当我们要求 `Date` 类找出两个日期之间的持续时间时，我们期望它能做“正确的事情”，所以我们必须确定什么是“正确的事情”。首先，空的日期在操作中是没有意义的，所有这样的尝试将导致异常的抛出。其次，当在操作中出现两个 `Date` 类对象时，如找这两个对象之间的持续时间，它们之间的“最小公分母”将决定结果。例如，下面的计算将返回 3 个月，因为代表 1997 年 1 月的 `d2`，对天数一无所知。

```
Date d1(1997,4,12), d2(1997,1);
```

```
int diff=d1-d2;           // 知道返回月
```

使 `Date` 类的方法包含所有的逻辑去处理在计算中部分日期可能出现的结合, 这样做当然是足够的了, 但是, 在类中对每个可能的地方都将逻辑概念进行封装通常是一个好主意。因此, 我要定义一个类的实现层次, 这样 `Date` 类可以根据需要做正确的事情, 如图 19.1 所示。

`Year` 类包含比较以及加和减年份的方法。一个 `YMonth` (代表“年和月”) 对象能在只涉及到年的地方代替 `Year`, 也可以在只涉及到月的地方代替月。

由于两个部分日期间的持续时间可以是年月日的任何组合, 因此我定义了一个 `Duration` 类去处理这样的返回值。按照第 13 章中的建议, 我定义了 `Date_Exception`——日期组件的异常类。

`Year` 类的定义在程序清单 19.8 和程序清单 19.9 中, 注意 `AddYears` 和 `SubtractYears` 的声明。它们是虚函数, 这是因为 `YMDay` 必须重载它们以正确地处理闰年。

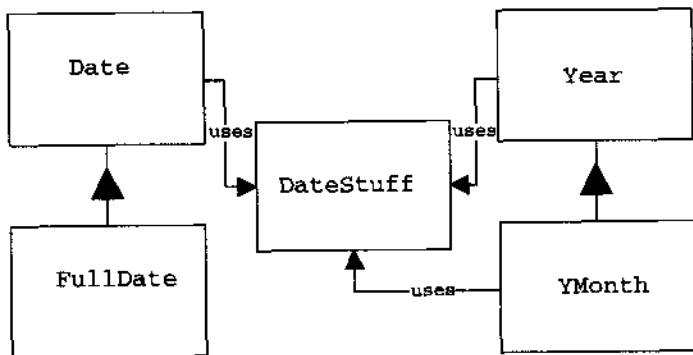


图 19.1 日期类的类框图

程序清单 19.8 Year 类的接口

```
// year.h
#ifndef YEAR_H
#define YEAR_H

#include "datestuff.h"

namespace DateStuff
{
    class Year
    {
    public:
        Year();           //当前的年
        Year(int);
        Year(const string&); //去掉开头的4个数字
        //存取器
        int GetYear() const;
    };
}
#endif
```

```

// 比较
int Compare(const Year& right) const;
bool operator==(const Year& right) const;
bool operator!=(const Year& right) const;
bool operator<(const Year& right) const;
bool operator>(const Year& right) const;
bool operator<=(const Year& right) const;
bool operator>=(const Year& right) const;

// 计算
int operator-(const Year& right) const; continued
// 其他操作
virtual void AddYears(int years);
virtual void SubtractYears(int years);
int YearsBetween(const Year& right) const;
Duration AgeBetween(const Year& right) const;
Duration Age() const;
int EndOfYear() const;

string ToString() const;
bool IsValid() const;
bool IsEmpty() const;

protected:
    int m_Year;
};

inline int Year::GetYear() const
{
    return m_Year;
}

inline int Year::Compare(const Year& right) const
{
    return m_Year - right.m_Year;
}

inline bool Year::operator==(const Year& right) const
{
    return Compare(right) == 0;
}

inline bool Year::operator!=(const Year& right) const
{
    return Compare(right) != 0;
}

```

```
    }

    inline bool Year::operator<(const Year& right) const
    {
        return Compare(right) < 0;
    }

    inline bool Year::operator>(const Year& right) const
    {
        return Compare(right) > 0;
    }
    inline bool Year::operator<=(const Year& right) const
    {
        return Compare(right) <= 0;
    }

    inline bool Year::operator>=(const Year& right) const
    {
        return Compare(right) >= 0;
    }

    inline int Year::operator-(const Year& right) const
    {
        return Compare(right);
    }

    inline Duration Year::AgeBetween(const Year& right) const
    {
        return Duration(YearsBetween(right));
    }

    inline int Year::YearsBetween(const Year& right) const
    {
        int diff = int(right.m_Year - m_Year);
        return int(diff >= 0 ? diff : -diff);
    }

    inline bool Year::IsValid() const
    {
        return MIN_YEAR <= m_Year && m_Year <= MAX_YEAR;
    }

    inline bool Year::IsEmpty() const
    {
        return m_Year == 0;
    }
```

```

inline string Year::ToString() const
{
    return IsEmpty() ? string()
        : DateStuff::ToString(m_Year,0,0);
}

} // Date 名字空间
#endif

```

程序清单 19.9 Year 类的实现

```

// year.cpp
#include "year.h"
#include <time.h>

using namespace DateStuff;

Year::Year()
{
    // 获得当前的年:
    time_t tval = time(0);
    struct tm *tmp = localtime(&tval);

    m_Year = int(tmp->tm_year + 1900);
}

Year::Year(int y)
{
    m_Year = y;
    if (!IsEmpty() && !IsValid())
        Throw(Date, YEAR_ERROR);
}

Year::Year(const string& s)
{
    m_Year = int(atoi(s.substr(0,4).c_str()));
    if (!IsEmpty() && !IsValid())
        Throw(Date, YEAR_ERROR);
}

void Year::AddYears(int y)
{
    if (m_Year > MAX_YEAR - y)
        Throw(Date, RANGE_ERROR);
    m_Year += y;
}

```

```
    }

    void Year::SubtractYears(int y)
    {
        if (m_Year < MIN_YEAR + y)
            Throw(Date, RANGE_ERROR);
        m_Year -= y;
    }

    Duration Year::Age() const
    {
        return AgeBetween(Year());
    }

    int Year::EndOfYear() const
    {
        if (!IsValid())
            Throw(Date, YEAR_ERROR);
        return DateStuff::EndOfYear(m_Year);
    }
}
```

YMonth(程序清单 19.10 和程序清单 19.11)增加了处理月份和季度的功能, 并且它的-运算符返回两个日期间的月份数。由于它与 Year 服从 is-a 的关系, 因此在 YMonth 把实现这个功能的责任交给了 Year (见函数 Compare 和函数 IsValid)。

程序清单 19.10 YMonth 类的接口

```
// ymonth.h
#ifdef YMONTH_H
#define YMONTH_H

#include "year.h"

namespace DateStuff
{
    class YMonth : public Year
    {
        typedef Year Super;

    public:
        YMonth();
        YMonth(int y, int m);
        YMonth(const string& s);

        //存取器
        int GetMonth() const;
    };
}
```

```

//比较
int Compare(const YMonth& right) const;
bool operator==(const YMonth& right) const;
bool operator!=(const YMonth& right) const;
bool operator<(const YMonth& right) const;
bool operator>(const YMonth& right) const;
bool operator<=(const YMonth& right) const;
bool operator>=(const YMonth& right) const;

//计算
long operator-(const YMonth& right) const;

// 其他操作
virtual void AddMonths(long months);
virtual void SubtractMonths(long months);
long MonthsBetween(const YMonth& right) const;
Duration AgeBetween(const YMonth& right) const;
Duration Age() const;
int EndOfYear() const;
int EndOfMonth() const;
int DaysInPrevMonth() const;
int DaysInNextMonth() const;

string ToString() const;
bool IsValid() const;
bool IsEmpty() const;

static Duration AgeBetween(const YMonth& first,
                           const YMonth& last);

protected:
    int m_Month;

private:
    void Resolve(long months);
};

inline int YMonth::GetMonth() const
{
    return m_Month;
}

inline int YMonth::Compare(const YMonth& right) const
{
    int diff = Super::Compare(right);
    return (diff == 0) ? m_Month - right.m_Month : diff;
}

```

```
}

inline bool YMonth::operator==(const YMonth& right) const
{
    return Compare(right) == 0;
}

inline bool YMonth::operator!=(const YMonth& right) const
{
    return Compare(right) != 0;
}

inline bool YMonth::operator<(const YMonth& right) const
{
    return Compare(right) < 0;
}

inline bool YMonth::operator>(const YMonth& right) const
{
    return Compare(right) > 0;
}

inline bool YMonth::operator<=(const YMonth& right) const
{
    return Compare(right) <= 0;
}

inline bool YMonth::operator>=(const YMonth& right) const
{
    return Compare(right) >= 0;
}

inline long YMonth::operator-(const YMonth& right) const
{
    return (m_Year - right.m_Year)*12L + m_Month - right.m_Month;
}

inline long YMonth::MonthsBetween(const YMonth& right) const
{
    long diff = (right - *this);
    return diff >= 0 ? diff : -diff;
}

inline void YMonth::AddMonths(long m)
{
    Resolve(m_Year*12L + m_Month + m);
}
```



```

}

inline void YMonth::SubtractMonths(long m)
{
    Resolve(m_Year*12L + m_Month - m);
}

inline Duration YMonth::AgeBetween(const YMonth& right) const
{
    return AgeBetween(right, *this);
}

inline bool YMonth::IsValid() const
{
    return Super::IsValid() &&
        1 <= m_Month && m_Month <= 12;
}

inline bool YMonth::IsEmpty() const
{
    return Super::IsEmpty() && m_Month == 0;
}

inline string YMonth::ToString() const
{
    char buf[23];    // 2 个整型数 + 1
    sprintf(buf, "%04d%02d", m_Year, m_Month);
    return IsEmpty() ? string() : string(buf);
}

inline void YMonth::Resolve(long months)
{
    DateStuff::ResolveMonths(months, m_Year, m_Month);
}

inline Duration YMonth::AgeBetween(const YMonth& p1,
                                   const YMonth& p2)
{
    return DateStuff::AgeBetween(p1.m_Year, p1.m_Month, 0,
                                 p2.m_Year, p2.m_Month, 0);
}

} // DateStuff 名字空间

#endif

```

程序清单 19.11 YMonth 类的实现

```
// ymonth.cpp
#include "ymonth.h"
#include <time.h>

using namespace DateStuff;

YMonth::YMonth()
{
    // 获得当前的年:
    time_t tval = time(0);
    struct tm *tmp = localtime(&tval);

    m_Month = int(tmp->tm_mon + 1);
}

YMonth::YMonth(int y, int m)
    : Year(y)
{
    m_Month = m;
    if (!IsEmpty() && !IsValid())
        Throw(Date, MONTH_ERROR);
}

YMonth::YMonth(const string& s)
    : Year(s)
{
    m_Month = int(atoi(s.substr(4,2).c_str()));
    if (!IsEmpty() && !IsValid())
        Throw(Date, MONTH_ERROR);
}

Duration YMonth::Age() const
{
    return AgeBetween(*this, YMonth());
}

int YMonth::EndOfYear() const
{
    if (!IsValid())
        Throw(Date, MONTH_ERROR);
    return DateStuff::EndOfYear(m_Year);
}
```

```

int YMonth::EndOfMonth() const
{
    if (!IsValid())
        Throw(Date, MONTH_ERROR);
    return DateStuff::EndOfMonth(m_Year, m_Month);
}

int YMonth::DaysInPrevMonth() const
{
    if (!IsValid())
        Throw(Date, MONTH_ERROR);
    return DateStuff::DaysInPrevMonth(m_Year, m_Month);
}

int YMonth::DaysInNextMonth() const
{
    if (!IsValid())
        Throw(Date, MONTH_ERROR);
    return DateStuff::DaysInNextMonth(m_Year, m_Month);
}

```

Year 类和对象 **Ymonth** 除了被 **Date** 和 **Full_Date** 类使用之外,在别的地方没有使用它们。如在程序清单 19.12 和程序清单 19.13 所见的那样, **Date** 函数决定了应用在所有实参上的最高精确度,然后按照需要把实际的工作委派给 **Year** 或 **YMonth**。例如, **AddDays** 函数知道只有带有有效的年月日的 **Date** 才能满足它的要求,所以它检查实参,并在 **YMDay** 不是有效值时抛出异常。**AgeBetween** 成员函数创建一个正确类型的临时对象,自动地调用合适的 **AgeBetween** 方法。**Date** 类同时提供字符串转换。这是很方便的,因为在数据库中我们实际上以 **YYYYMMDD** 格式的字符串存储日期,在那里日、月和日或是全部的三部分都可以缺少。在程序清单 19.14 和程序清单 19.15 里, **FullDate** 是 **Date** 的一个特例,它要求在所有实例中都有有效的年月日。因为设计类的方法, **FullDate** 只需要在其构造函数中加强这点,其他的部分则可以自己解决。在程序清单 19.16 中说明了 **Date** 和 **FullDate** 类的功能。

程序清单 19.12 **Date** 类的接口

```

// date.h
#ifndef DATE_H
#define DATE_H

#include "datestuff.h"
#include "year.h"
#include "ymonth.h"

namespace DateStuff
{

```

```
class Date
{
public:
    Date(int = 0, int = 0, int = 0);
    Date(const string&);
    Date(const Year&);
    Date(const YMonth&);

    //附加的赋值(为了提高效率):
    Date& operator=(const string&);
    Date& operator=(const Year&);
    Date& operator=(const YMonth&);

    //存取器
    int GetYear() const;
    int GetMonth() const;
    int GetDay() const;

    // 测试:
    bool IsValidYear() const;
    bool IsValidYMonth() const;
    bool IsValidYMDay() const;
    virtual bool IsValid() const; //上面 3 个中的一个
    bool IsEmpty() const;

    // 比较:
    int Compare(const Date&) const;
    bool operator==(const Date&) const;
    bool operator!=(const Date&) const;
    bool operator<(const Date&) const;
    bool operator>(const Date&) const;
    bool operator<=(const Date&) const;
    bool operator>=(const Date&) const;

    // 基本计算:
    void AddYears(int);
    void SubtractYears(int);
    int YearsBetween(const Date&);
    void AddMonths(long);
    void SubtractMonths(long);
    long MonthsBetween(const Date&) const;
    void AddDays(long);
    void SubtractDays(long);
    long DaysBetween(const Date&) const;
    void AddWeeks(long weeks);
```

```

void SubtractWeeks(long weeks);
long WeeksBetween(const Date& const;
void AddWeekDays(long days);
void SubtractWeekDays(long days);
long WeekDaysBetween(const Date& right) const;
Duration operator-(const Date&) const;

// 特殊请求:
int DayOfWeek() const;
int DayOfYear() const;
int EndOfMonth() const;
int EndOfYear() const;
Duration AgeBetween(const Date&) const;
Duration Age() const;
static Date Today();

// 变换:
string ToString() const;

protected:
    int m_Year;
    int m_Month;
    int m_Day;
};

inline Date::Date(const Year& y)
{
    m_Year = y.GetYear();
    m_Month = 0;
    m_Day = 0;
}

inline Date::Date(const YMonth& ym)
{
    m_Year = ym.GetYear();
    m_Month = ym.GetMonth();
    m_Day = 0;
}

inline Date& Date::operator=(const Year& y)
{
    assert(y.IsEmpty() || y.IsValid());
    m_Year = y.GetYear();
    m_Month = 0;
    m_Day = 0;
    return *this;
}

```

```
    }

    inline Date& Date::operator=(const YMonth& ym)
    {
        assert(ym.IsEmpty() || ym.IsValid());
        m_Year = ym.GetYear();
        m_Month = ym.GetMonth();
        m_Day = 0;
        return *this;
    }

    inline int Date::GetYear() const
    {
        return m_Year;
    }

    inline int Date::GetMonth() const
    {
        return m_Month;
    }

    inline int Date::GetDay() const
    {
        return m_Day;
    }

    inline bool Date::IsValidYear() const
    {
        return DateStuff::IsValidYear(m_Year);
    }

    inline bool Date::IsValidYMonth() const
    {
        return DateStuff::IsValidYMonth(m_Year, m_Month);
    }

    inline bool Date::IsValidYMDay() const
    {
        return DateStuff::IsValidYMDay(m_Year, m_Month, m_Day);
    }

    inline bool Date::IsValid() const
    {
        return IsEmpty() ||
            (m_Day == 0 && m_Month == 0 && IsValidYear()) ||
            (m_Day == 0 && IsValidYMonth()) ||
            IsValidYMDay();
    }
```

```
}

inline bool Date::IsEmpty() const
{
    return m_Day == 0 && m_Month == 0 && m_Year == 0;
}

inline int Date::Compare(const Date& r) const
{
    int ydiff = m_Year - r.m_Year;
    int mdiff = m_Month - r.m_Month;
    return (ydiff == 0) ? ((mdiff == 0) ? m_Day - r.m_Day
                                     : mdiff)
        : ydiff;
}

inline bool Date::operator==(const Date& r) const
{
    return Compare(r) == 0;
}

inline bool Date::operator!=(const Date& r) const
{
    return Compare(r) != 0;
}

inline bool Date::operator<(const Date& r) const
{
    return Compare(r) < 0;
}

inline bool Date::operator>(const Date& r) const
{
    return Compare(r) > 0;
}

inline bool Date::operator<=(const Date& r) const
{
    return Compare(r) <= 0;
}

inline bool Date::operator>=(const Date& r) const
{
    return Compare(r) >= 0;
}
```

```
inline Duration Date::operator-(const Date& r) const
{
    return AgeBetween(r);
}

inline string Date::ToString() const
{
    return IsEmpty() ? string()
        : DateStuff::ToString(m_Year, m_Month, m_Day);
}

inline Date Date::Today()
{
    int y, m, d;
    DateStuff::Today(y,m,d);
    return Date(y,m,d);
}

} //结束 DateStuff 命名空间
#endif
```

程序清单 19.13 Date 类的实现

```
// date.cpp
#include "date.h"

using namespace DateStuff;

Date::Date(int y, int m, int d)
{
    m_Year = y;
    m_Month = m;
    m_Day = d;
    if (!IsEmpty() && !IsValid())
        Throw(Date,DATE_ERROR);
}

Date::Date(const string& s)
{
    // 需要 YYYYMMDD
    if (s.size() > 8)
        Throw2(Date,DATE_ERROR,"String too long");

    m_Year = int(atoi(s.substr(0,4).c_str()));
    m_Month = int(atoi(s.substr(4,2).c_str()));
    m_Day = int(atoi(s.substr(6,2).c_str()));
}
```



```

        if (!IsEmpty() && !IsValid())
            Throw(Date, DATE_ERROR);
    }

Date& Date::operator=(const string& s)
{
    // 需要 YYYYMMDD 剩下的子集
    if (s.size() > 8)
        Throw2(Date, DATE_ERROR, "String too long");

    m_Year = int(atoi(s.substr(0,4).c_str()));
    m_Month = int(atoi(s.substr(4,2).c_str()));
    m_Day = int(atoi(s.substr(6,2).c_str()));
    if (!IsEmpty() && !IsValid())
        Throw(Date, DATE_ERROR);
    return *this;
}

void Date::AddYears(int years)
{
    if (IsValidYMDay())
    {
        DateStuff::AddYears(years, m_Year, m_Month, m_Day);
    } continued
    else if (IsValidYMonth() || IsValidYear())
    {
        Year y(m_Year);
        y.AddYears(years);
        operator=(y);
    }
    else
        Throw(Date, DATE_ERROR);
}

void Date::SubtractYears(int years)
{
    if (IsValidYMDay())
    {
        DateStuff::SubtractYears(years, m_Year, m_Month, m_Day);
    }

    else if (IsValidYMonth() || IsValidYear())
    {
        Year y(m_Year);
        y.SubtractYears(years);
        operator=(y);
    }
}

```

```
    }
    else
        Throw(Date, DATE_ERROR);
}

int Date::YearsBetween(const Date& r)
{
    if (!IsValid())
        Throw(Date, DATE_ERROR);
    Year me(m_Year);
    Year you(r.m_Year);
    return me.YearsBetween(you);
}

void Date::AddMonths(long months)
{
    if (IsValidYMDay())
    {
        DateStuff::AddMonths(months, m_Year, m_Month, m_Day);
    }
    else if (IsValidYMonth())
    {
        DateStuff::ResolveMonths(m_Year*12L + m_Month + months,
                                  m_Year, m_Month);
    }
    else
        Throw(Date, DATE_ERROR);
}

void Date::SubtractMonths(long months)
{
    if (IsValidYMDay())
    {
        DateStuff::SubtractMonths(months, m_Year, m_Month, m_Day);
    }
    else if (IsValidYMonth())
    {
        DateStuff::ResolveMonths(m_Year*12L + m_Month - months,
                                  m_Year, m_Month);
    }
    else
        Throw(Date, DATE_ERROR);
}

long Date::MonthsBetween(const Date& r) const
{

```

```

        if (!IsValidYMonth() || !r.IsValidYMonth())
            Throw(Date, DATE_ERROR);
        return DateStuff::MonthsBetween(m_Year, m_Month, r.m_Year, r.m_Month);
    }

    void Date::AddDays(long days)
    {
        if (!IsValidYMDay())
            Throw(Date, DATE_ERROR);
        DateStuff::AddDays(days, m_Year, m_Month, m_Day);
    }

    void Date::SubtractDays(long days)
    {
        if (!IsValidYMDay())
            Throw(Date, DATE_ERROR);
        DateStuff::SubtractDays(days, m_Year, m_Month, m_Day);
    }

    long Date::DaysBetween(const Date& r) const
    {
        if (!IsValidYMDay() || !r.IsValidYMDay())
            Throw(Date, DATE_ERROR);
        return DateStuff::DaysBetween(m_Year, m_Month, m_Day,
                                       r.m_Year, r.m_Month, r.m_Day);
    }

    void Date::AddWeekDays(long days)
    {
        if (!IsValidYMDay())
            Throw(Date, DATE_ERROR);
        DateStuff::AddWeekDays(days, m_Year, m_Month, m_Day);
    }

    void Date::SubtractWeekDays(long days)
    {
        if (!IsValidYMDay())
            Throw(Date, DATE_ERROR);
        DateStuff::SubtractWeekDays(days, m_Year, m_Month, m_Day);
    }

    long Date::WeekDaysBetween(const Date& r) const
    {
        if (!IsValidYMDay() || !r.IsValidYMDay())
            Throw(Date, DATE_ERROR);
        return DateStuff::WeekDaysBetween(m_Year, m_Month, m_Day,

```

```
        r.m_Year, r.m_Month, r.m_Day);
    }

    void Date::AddWeeks(long weeks)
    {
        if (!IsValidYMDay())
            Throw(Date, DATE_ERROR);
        DateStuff::AddWeeks(weeks, m_Year, m_Month, m_Day);
    }

    void Date::SubtractWeeks(long weeks)
    {
        if (!IsValidYMDay())
            Throw(Date, DATE_ERROR);
        DateStuff::SubtractWeeks(weeks, m_Year, m_Month, m_Day);
    }

    long Date::WeeksBetween(const Date& r) const
    {
        if (!IsValidYMDay() || !r.IsValidYMDay())
            Throw(Date, DATE_ERROR);
        return DateStuff::WeeksBetween(m_Year, m_Month, m_Day,
                                         r.m_Year, r.m_Month, r.m_Day);
    }

    Duration Date::AgeBetween(const Date& r) const
    {
        if (!IsValid() || !r.IsValid())
            Throw(Date, DATE_ERROR);

        if (IsValidYMDay() && r.IsValidYMDay())
            return DateStuff::AgeBetween(m_Year, m_Month, m_Day,
                                         r.m_Year, r.m_Month, r.m_Day);
        else if (IsValidYMonth() && r.IsValidYMonth())
            return DateStuff::AgeBetween(m_Year, m_Month, 0,
                                         r.m_Year, r.m_Month, 0);
        else if (IsValidYear() && r.IsValidYear())
            return DateStuff::AgeBetween(m_Year, 0, 0, r.m_Year, 0, 0);
        else
        {
            // 不应在这里获得:
            assert(0);
            return Duration(-1, -1, -1); // MS 需要明确的返回!
        }
    }
}
```

```

Duration Date::Age() const
{
    return AgeBetween(Today());
}

int Date::DayOfWeek() const
{
    if (!IsValidYMDay())
        Throw(Date, DATE_ERROR);
    return DateStuff::DayOfWeek(m_Year, m_Month, m_Day);
}

int Date::DayOfYear() const
{
    if (!IsValidYMDay())
        Throw(Date, DATE_ERROR);
    return DateStuff::DayOfYear(m_Year, m_Month, m_Day);
}

int Date::EndOfMonth() const
{
    if (!IsValidYMonth())
        Throw(Date, DATE_ERROR);
    return DateStuff::EndOfMonth(m_Year, m_Month);
}

int Date::EndOfYear() const
{
    if (!IsValidYear())
        Throw(Date, DATE_ERROR);
    return DateStuff::EndOfYear(m_Year);
}

```

程序清单 19.14 FullDate 类的接口

```

// FullDate.h
#ifndef FULLDATE_H
#define FULLDATE_H

#include "date.h"

namespace DateStuff
{
    class FullDate : public Date
    {

```

```
public:
    FullDate();
    FullDate(const string&);
    FullDate(int, int, int);
    FullDate(const Date&);

    bool IsValid() const;
    long ToJul() const;
};

inline FullDate::FullDate()
{
    DateStuff::Today(m_Year, m_Month, m_Day);
}

inline bool FullDate::IsValid() const
{
    return IsEmpty() || DateStuff::IsValidYMDay(m_Year, m_Month, m_Day);
}

inline long FullDate::ToJul() const
{
    return GregToJul(m_Year, m_Month, m_Day);
}

} //结束 DateStuff 命名空间
#endif
```

程序清单 19.15 FullDate 类的实现

```
// fulldate.cpp
#include "fulldate.h"

using namespace DateStuff;

FullDate::FullDate(const string& s)
{
    DateStuff::FromString(s, m_Year, m_Month, m_Day);
    if (!IsEmpty() && !IsValidYMDay())
        Throw(Date, DATE_ERROR);
}

FullDate::FullDate(int year, int month, int day)
{
    m_Year = year;
    m_Month = month;
```

```

    m_Day = day;
    if (!IsEmpty() && !IsValidYMDay())
        Throw(Date, DATE_ERROR);
}

```

```

FullDate::FullDate(const Date& d)
{
    m_Year = d.GetYear();
    m_Month = d.GetMonth();
    m_Day = d.GetDay();
    assert(IsValid());
}

```

程序清单 19.16 说明 Date 和 FullDate 功能

```

// tdate.cpp: 测试 Date 和 FullDate
#include <iostream>
#include <stdexcept>
#include "fulldate.h"

using namespace DateStuff;
using namespace std;

void checkDate(int, int, int);
const char* DayText[] = {"Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday", "Sunday"};
main()
{
    checkDate(0,3,14);      //无效的年
    checkDate(1997,0,12);   //无效的月
    checkDate(1998,13,1);   //无效的月
    checkDate(97,1,12);     //无效的年
    checkDate(1999,2,29);   //无效的天
    checkDate(1900,2,29);   //无效的天
    Date empty(0,0,0);
    cout << "empty: \"" << empty.ToString() << "\"\n";
    Date empty2("");
    cout << "empty2: \"" << empty2.ToString() << "\"\n";

    Date d1 = string("20000101");
    Date d2(2000,3,1);
    cout << "Day of week for d1: " << d1.ToString() << " == "
        << DayText[d1.DayOfWeek()] << endl;
    cout << "Day of week for d2: " << d2.ToString() << " == "
        << DayText[d2.DayOfWeek()] << endl;
}

```

```
Date p1(1996,2,29);
Date p2("20000229");

cout << "p1 == " << p1.ToString() << endl;
cout << "p2 == " << p2.ToString() << endl;

Duration d = p1.AgeBetween(p2);
cout << "Duration between p1 and p2: "
    << d.year << " years, "
    << d.month << " months, "
    << d.day << " days\n";

Duration dd = p1.AgeBetween(d2);
cout << "Duration between p1 and d2 (20000301): "
    << dd.year << " years, "
    << dd.month << " months, "
    << dd.day << " days\n";

if (p1 < p2)
{
    p1.AddYears(d.day);
    p1.AddMonths(d.month);
    p1.AddYears(d.year);
}
else if (p1 > p2)
{
    p2.AddYears(d.day);
    p2.AddMonths(d.month);
    p2.AddYears(d.year);
}

cout << "p1 == p2 (after adding Duration(p1,p2)): "
    << (p1 == p2 ? "Yes" : "No")
    << endl;

FullDate today;
Date f(today);
cout << "Today: " << f.ToString() << endl;
f.AddWeekDays(2);
cout << "+2 weekdays: " << f.ToString() << endl;
f.AddWeekDays(10);
cout << "+10 weekdays: " << f.ToString() << endl;
f.AddWeekDays(30);
cout << "+30 weekdays: " << f.ToString() << endl;
f.SubtractWeekDays(30);
cout << "-30 weekdays: " << f.ToString() << endl;
```



```

f.SubtractWeekDays(10);
cout << "-10 weekdays: " << f.ToString() << endl;
f.SubtractWeekDays(2);
cout << "-2 weekdays: " << f.ToString() << endl;

cout << "WeekdaysBetween 12/11/96 and 12/13/96: "
    << Date(1996,12,11).WeekDaysBetween(Date(1996,12,13))
    << endl;
cout << "WeekdaysBetween 12/13/96 and 12/27/96: "
    << Date(1996,12,13).WeekDaysBetween(Date(1996,12,27))
    << endl;
cout << "WeekdaysBetween 12/27/96 and 2/7/97: "
    << Date(1996,12,27).WeekDaysBetween(Date(1997,2,7))
    << endl;

FullDate f1(d1), f2(d2);
Duration f12 = f2 - f1;
cout << "d2 - d1 == "
    << f12.year << " years, "
    << f12.month << " months, "
    << f12.day << " days\n";
}

void checkDate(int year, int month, int day)
{
    try
    {
        Date(year,month,day);
        cout << "Validity check FAILED" << endl;
    }
    catch (Date_Exception& x)
    {
        cout << "Validity check PASSED (" << x.what() << ") " << endl;
    }
}

// 输出:
Validity check PASSED (Invalid Date:DATE.CPP:Line 15)
Validity check PASSED (Invalid Date:DATE.CPP:Line 15)
Validity check PASSED (Invalid Date:DATE.CPP:Line 15)
Validity check PASSED (Invalid Date:DATE.CPP:Line 15)
Validity check PASSED (Invalid Date:DATE.CPP:Line 15)
Validity check PASSED (Invalid Date:DATE.CPP:Line 15)
empty: ""
empty2: ""
Day of week for d1: 20000101 == Saturday
Day of week for d2: 20000301 == Wednesday

```

```

p1 == 19960229
p2 == 20000229
Duration between p1 and p2: 4 years, 0 months, 0 days
Duration between p1 and d2 (20000301): 4 years, 0 months, 1 days
p1 == p2 (after adding Duration(p1,p2)): Yes
Today: 19970412
+2 weekdays: 19970415
+10 weekdays: 19970429
+30 weekdays: 19970610
-30 weekdays: 19970429
-10 weekdays: 19970415
-2 weekdays: 19970411
WeekdaysBetween 12/11/96 and 12/13/96: 2
WeekdaysBetween 12/13/96 and 12/27/96: 10
WeekdaysBetween 12/27/96 and 2/7/97: 30
d2 - d1 == 0 years, 2 months, 0 days

```

19.3 计算年的星期数

一年（或月）的绝对星期是以七天制为基础的，它开始于第一天，结束于第七天，这就是第一周。因此通过简单的公式可以查找天数的绝对星期数：

$$w = (d - 1) / 7 + 1;$$

其中 d 是天数。它对计算年的星期数和月的星期数都起作用。

普通星期是以从星期日到星期六的持续时间为基础。年或月的第一个普通星期开始于第一天，在所遇到的第一个星期六结束（它们也可能就是同一天）。中间的一周是从星期日到星期六，最后一周开始于最后一个星期日，结束于最后一天（它又有可能就是这周开始的星期日）。因此，一年中有 53 个普通星期（不包括闰年的第一个星期六是新年的情况，在这种情况下有 54 个普通星期），并且一个月中有 5 个普通星期（不包括 2 月有 28 天并且开始于星期日的情况，在这种情况下有 4 个普通星期）。

给出日的序数为了计算年的普通星期，注意下面的格式（ s 代表该年的第一个星期六的序数）：

星期数	开始天	结束天
1	1	s
2	$s+1$	$s+7$
3	$s+8$	$s+14$
n	$s+1+7(n-2)$	$s+7(n-1)$

因为 $n \geq 2$ ，那么可以写出：

$$s + 1 + 7 (n - 2) \leq d \leq s + 7 (n - 1)$$

其中, d 是正在讨论的那一天。解出这个不等式中的 n :

$$1 + (d - s) / 7 \leq n \leq 2 + (d - s - 1) / 7 \quad // \text{整数运算}$$

当 $d - s$ 是 7 的倍数时不等式取等, 即当 d 是星期六的时候。当 d 不是星期六时, 不等式两端的值相差 1。问题是这两个值中哪一个是 n 的正确值? 试验一些日期的结果表明上限是 n 的正确值 (计算机模拟验证了这点)。它与少于 7 天的日期在第二普通的星期里这个事实是一致的 (不可能和下限一致)。一个日期 d 的普通星期因此可由下面的简单公式决定:

$$n = 2 + (d - s - 1) / 7$$

其中, d 和 s 的意思同上。这个公式也支持计算月的普通星期, 此时 d 是月中的某一天, s 是其第一个星期六的序数。

一年中普通星期数, 正如上面提到的, 虽然通常是 53, 但是偶然也有 54 的情况, 可以通过设置上面表中公式的下限等于一年的最后一天来获得 (e 等于 365 或是 366)

$$s + 1 + 7 (n - 2) = e$$

并解出 n :

$$n = (e - s - 1) / 7 + 2$$

19.4 小结

- 利用 `<time.h>` 中的函数, 如 `localtime`、`strftime` 和 `difftime`, 进行简单的时间和日期操作。
- 确信知道环境的参考日期。
- 以整型为基础的 Julian 日期编码便于一个大范围内日期的简单有效的计算 (即在 1582 年 10 月 15 日之后)。
- `Date` 和 `FullDate` 类处理最常见的商业日期处理需求, 包括部分日期。

19.5 参考文献

Burki, David, *Date Conversion*, *C User Journal*, Feb. 1993, pp. 29-34.

动态内存管理

作为一个 C/C++ 的程序员需要考虑程序对象的三种内部的存储方式：自动存储、静态存储和动态存储。当在程序块中定义自动的变量时，大部分编译器为这些变量在程序堆栈或寄存器里分配空间。当执行离开程序块，堆栈指针返回到它进入程序块之前的地方，有效地销毁了那个程序块的自动变量。当执行重新进入该块时，它用和以前一样的初始值再次创建所有的自动变量。静态对象的声明或者在文件域中，或者在有静态说明符的命名空间里。它们在程序的数据空间的一个固定的位置上保存并且在第一次使用之前被初始化一次。动态对象由特殊的系统调用在运行期中创建，并保存在堆（或自由存储区）中，这是一个特殊的留给用户控制的数据区，用于运行期内存分配。当为一个对象申请堆栈空间时，可在返回时得到一个地址。系统为你保留了这个地址对应的内存直到你释放它为止。在本章中讨论了在 C/C++ 中所使用的动态内存的常见技术。

20.1 参差数组

在编译期间当不知道需要多大的空间来存储一个对象或者不知道将需要多少个对象的时候，使用堆栈是很方便的。例如，在 C 中处理文本文件的习惯做法是为文本的每一行分配堆栈空间，并在一个指向 char 型的指针数组里存储地址。程序清单 20.1 中的程序把文本的 MAXLINES 行读入内存并使用 qsort 按升序给它们分类。（关于 qsort 函数的介绍参见第 15 章）当程序读每一行的时候，它调用 malloc（一个 <stdlib.h> 函数）得到足够的堆栈空间来保存该行及其 null 结束字节。它把 malloc 返回的地址存储在数组 strings 中。这个灵活的存储机制有时叫做参差数组，因为它只分配文本每一行所需的内存，如下表所描述的那样。

用于读和分类的行	参差数组 srtings[]
srtings[0]	now\0
srtings[1]	is\0
srtings[2]	the\0
srtings[3]	time\0
等等	

程序清单 20.1 一个分类程序

```

/* sort.c: 分类字符串 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#define MAXLINES 1024

static int scomp(const void *, const void *);

main(int argc, char *argv[])
{
    int i, n;
    char *strings[MAXLINES], buf[BUFSIZ];

    if (argc > 1)
        freopen(argv[1], "r", stdin);

    for (n = 0; n < MAXLINES && fgets(buf, BUFSIZ, stdin); ++n)
    {
        strings[n] = malloc(strlen(buf)+1);
        assert(strings[n]);
        strcpy(strings[n], buf);
    }

    qsort(strings, n, sizeof strings[0], scomp);

    for (i = 0; i < n; ++i)
    {
        fputs(strings[i], stdout);
        free(strings[i]);
    }
    return 0;
}

static int scomp(const void *p1, const void *p2)
{

```

```

    char *a = * (char **) p1;
    char *b = * (char **) p2;
    return strcmp(a,b);
}

```

尽管每一行都有一个指针是一个巨大的开支，但是这样可能比使用一个行的长度非常大的二维存储机制更有效率。

用于读和分类的行	参差的 syrnigs[] 数组	二维存储 trings[][MAXLINE]
syrnigs[0]	now\0	now\0 ? ? ? ?
syrnigs[1]	is\0	is\0 ? ? ? ? ?
syrnigs[2]	the\0	the \0 ? ? ? ?
syrnigs[3]	time\0	time \0 ? ? ?
Etc.		

当释放动态内存时，应该把这个内存的地址作为参数传递给 free 函数，该函数使这个内存空间在将来调用 malloc 函数时可被利用。

20.2 在标准 C 中使用堆

头文件<stdlib.h>为使用动态内存而声明了四个函数：

1. void *malloc(size_t siz);

返回一个指向 siz 的第一个字节的指针。通常用于分配单个对象。

2. void *calloc(size_t nelems, size_t elem_size);

返回指向 nelems*elem_size 字节的指针，并被初始化为 0。通常用于分配对象数组。

3. void *realloc(void *ptr, size_t siz);

用于扩展或缩小堆栈的分配。除非 ptr 是 NULL 指针，否则 ptr 指针必须源于上次对 malloc、calloc 或 realloc 的调用，其结果和 malloc (siz) 相同。如果有足够的空间用于新的分配，realloc 就把原始数据拷贝到新分配的内存中，然后返回这个新的地址。如果 ptr 不是 NULL，而 siz 是 0 时，那么 realloc 就和 free 的作用一样了。

4. void free(void *ptr);

使以前分配的堆栈内存区可以重用。前一次的调用一定会返回 ptr 指向口述之一的内存分配函数。调用 free (NULL) 是合法的，但它不起任何作用。

如果被请求分配的内存区不可用，这三个内存分配函数就返回一个 NULL 指针。

程序清单 20.2 和 20.3 中的程序说明了所有的四个函数。它通过允许指定参数文件的方法来为传统的 argc/argv 机制扩展了读取命令行参数的能力。程序中任何以“@”字符开头的参数都命名了一个间接文件，这个文件包含了更多的参数。例如，命令行：

```
getargs @arg.dat
```

使 arg.dat 被用于读取更多的参数。像这样的命令文件可以嵌套（见程序清单 20.4）。

程序清单 20.2 举例说明间接文件

```

/* getargs.c: 读参数文件 */
#include <stdio.h>

extern char **arglist(int,char **,int *);
extern void free_arglist(int,char **);

main(int argc, char *argv[])
{
    int i, nargs;
    char **args = arglist(--argc,++argv,&nargs);

    for (i = 0; i < nargs; ++i)
        printf("%d: %s\n",i,args[i]);
    free_arglist(nargs,args);
    return 0;
}

/* 样本执行:
c:> getargs @arg.dat
0: little
1: lamb
2: where
3: no
4: one
5: along
6: came
7: a
8: spider
9: has
10: gone
11: before
12: little
13: lamb
*/

```

`arglist` 函数返回一个指针，它指向一个动态分配大小的、字符串的参差数组，这个数组是一个完全扩展的程序参数列表。函数 `free_args` 释放所有在创建数组时使用的内存。

函数 `arglist` 使用 `calloc` 函数来动态地分配一个足够大的数组来容纳 `argc-1` 个参数，也就是命令行中参数的原始个数。它调用 `add` 函数来插入一个新的参数到列表中。如果数组是满的，`add` 函数就调用 `realloc` 函数通过一个预先指定的数量 (`CHUNK`) 来增加数组的长度。`add` 函数调用 `malloc` 把所有的字符串参数放到堆中。函数 `expand` 用于处理间接文件参数。由于间接文件可以被嵌套，所以函数 `expand` 是一个递归函数。如果没有间接文件参数，那么 `realloc` 函数就永远不会被调用。

程序清单 20.3 从间接文件中读取参数的函数

```
/* arglist.c: 从文件递归地读取参数 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#define CHUNK 10      /* 重新分配的数量 */
static char **args;   /* 参数列表 */
static int nleft;     /* 不用的参数位置 */
static int nargs;     /* 参数的数量 */
/* 私有函数 */
static void expand(FILE *f);
static void add(char *arg);

char **arglist(int old_nargs, char **old_args, int *new_nargs)
{
    int i;

    /* 最初的分配 */
    args = calloc(old_nargs, sizeof(char *));
    assert(args);
    nleft = old_nargs;
    nargs = 0;

    /* 处理每个命令行的参数 */
    for (i = 0; i < old_nargs; ++i)
        if (old_args[i][0] == '@')
        {
            /* 打开参数文件 */
            FILE *f = fopen(old_args[i]+1, "r");
            if (f)
            {
                expand(f);
                fclose(f);
            }
        }
        else
            add(old_args[i]);

    *new_nargs = nargs;
    return args;
}

void free_arglist(int n, char **av)
{

```

```
    int i;
    for (i = 0; i < n; ++i)
        free(av[i]);
    free(av);
}

static void expand(FILE *f)
{
    char token[BUFSIZ];

    while (fscanf(f,"%s",token) == 1)
        if (token[0] == '@')
        {
            FILE *g = fopen(token+1,"r");
            if (g)
            {
                expand(g);
                fclose(g);
            }
        }
        else
            add(token);
}

static void add(char *arg)
{
    if (nleft == 0)
    {
        /* 扩大参数列表 */
        args = realloc(args,(nargs+CHUNK) * sizeof(char *));
        assert(args);
        nleft = CHUNK;
    }

    /* 为存储参数分配空间 */
    args[nargs] = malloc(strlen(arg) + 1);
    assert(args[nargs]);
    strcpy(args[nargs++], arg);
    --nleft;
}
```

程序清单 20.4 输入文件来检测参数列表函数

```
File arg.dat:
little lamb
@arg2.dat
little
```

```

lamb
File arg2.dat:
where no one
@arg3.dat
has gone
before
File arg3.dat:
along came
a
spider

```

20.3 C++的自由存储

作为<stdlib.h>中内存管理函数的替代，C++提供了 `new` 和 `delete` 运算符。`operator` 是关键字。在 C 中必须显式地把操作对象的大小传递给 `malloc` 函数。由于 `new` 是 C++ 中的运算符，所以编译器计算出了用于处理对象所需要的内存大小。如果要得到一个指向任意类型的动态对象的指针，只需这样向系统发出请求即可：

```
T*tp=new T;
```

使用 `new` 可以做两件事：

1. 为对象分配所需的内存；
2. 调用适当的构造函数。

数组也很简单：

```
const size_t SIZE =100;
T*tap=new T(SIZE);
```

在这种情况下 `new` 为了增加下标值而调用了默认的构造函数来初始化每一个数组元素。

即使是多维数组也很简单。例如，要分配一个两行三列的数组，可以这样做：

```
const size_t NROWS=2;
const size_t NCOLS=3;
T (*p) (NCOLS) =new T[NROWS][NCOLS];
```

现在可以用数组语法来使用 `p`，如下：

```
T t;
```

```
...
```

```
p[0][1]=t;
```

（如果对 `p` 的声明语法还感到困惑，请参阅第 2 章）。

要删除堆对象，可以使用 `delete` 运算符：

```
delete tp; //数量形式
delete [] tap; // 数组形式
```

在回收内存之前，调用 `delete` 也就调用了合适的析构函数。数组形式 `delete[]`，通过减小数组下标值的顺序来删除数组元素。当向堆返回内存时，使用正确的 `delete` 形式是很重要的。

20.4 浅拷贝与深拷贝

在程序清单 20.5 和 20.6 中简单的字符串类使用了 C 风格的字符串来保存它的数据。由于字符串可以增长和缩短，因此数据缓存区在堆中分配。然而，程序清单 20.7 中的测试程序暴露了使用这个类所存在的问题。由于某种原因，一个字符串，无论是从另一个字符串初始化而来的还是分配而来的，总是保持和这个原始的字符串“连接”，以致改变了一个字符串就改变了另一个。如果仔细地查看这个类的定义，就会注意到我没有定义一个拷贝构造函数或者一个赋值运算符。一个拷贝构造函数有以下的格式：

```
String(const String&);
```

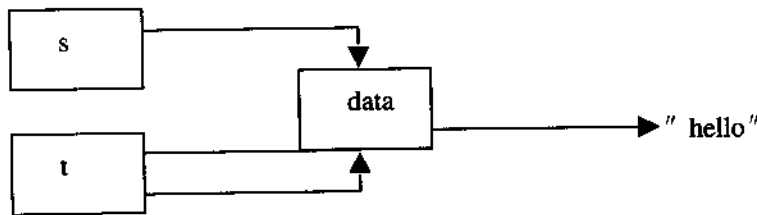
并且，每当一个新的 String 对象用另一个 String 对象的值初始化时，该拷贝构造函数都会执行，如下所示：

```
String t=s;
```

赋值运算符有以下的格式：

```
String& operator=(const String&);
```

如果没这么做，编译器也会自动地生成这些函数。然而，这些自动的形式使用的是按成员语法。这意味着当使用一个字符串对象的值去初始化一个新的字符串对象时，接收对象的每一个成员都会用原始对象中的相应成员的值来被初始化。如果数据成员是一个具有拷贝构造函数类型的实例，那么该拷贝构造函数就被调用。就 String 类来说，数据成员是内置类型，因此指针和计数器只是以标量值的形式被拷贝。结果如下：



程序清单 20.5 简单 string 类的头文件

```
// str.h
#include <stddef.h>

class ostream;

class String
{
public:
    String(const char*);
    ~String();
```

```

    char& operator[](size_t pos);
    friend ostream& operator<<(ostream&, const String&);

private:
    char* data;
    size_t count;
    void clone(const char *);
};

inline String::String(const char *s)
{
    clone(s);
}

inline String::~String()
{
    delete [] data;
}

inline char& String::operator[](size_t pos)
{
    return data[pos];
}

```

程序清单 20.6 string 类的实现文件

```

// str.cpp
#include <iostream>
#include <string.h>
#include "str.h"

ostream& operator<<(ostream& os, const String& s)
{
    os.write(s.data,s.count);
    return os;
}

void String::clone(const char *s)
{
    count = strlen(s);
    data = new char[count + 1];
    strcpy(data,s);
}

```

程序清单 20.7 显示 string 类的浅拷贝

```

// tstr.cpp:    测试 C++ 字符串类
#include <iostream>

```

```

#include "str.h"

main()
{
    String s = "hello", t = s;

    t[0] = 'j';
    cout << "s == " << s << endl;
    cout << "t == " << t << endl;
}

// 输出:
s == jello
t == jello

```

由于编译器产生的拷贝构造函数时只把 `s.data` 拷贝到 `t.data` 中, 因此字符串对象 `s` 和 `t` 实际上共享了基本的 C 字符串缓存区。这就被称为浅拷贝。如果 `s` 应该在 `t` 之前离开生存空间, 那么 `String` 类的析构函数将使 `t.data` 成为一个虚指针:



用户真正想要得到的是一个深拷贝, 这种深拷贝会分别为每一次初始化或赋值创建一个单独的基本字符串缓存区拷贝。这就需要有一个拷贝构造函数和一个赋值运算符来为接收方对象分配新的缓存区, 如下:

```

String(const String& s);
{
    clone(s.data);
}
String& String::operator=(const String& s);
{
    if (this != &s)
    {
        delete[] data;
        clone(s.data);
    }
    return *this;
}

```

一般来说, 当一个对象包含了一个指向堆内存的指针时, 应该定义一个拷贝构造函数和赋值运算符, 以及所需的析构函数和其他的构造函数。

20.5 处理内存分配失败

在开发 C++ 的异常处理部分之前，如果一个 `new` 操作分配对象失败，则它会返回一个空指针，就像 `malloc` 在 C 中所做的一样：

```
T *tp = new T;
if (tp)
    //使用 new 对象
```

现在 C++ 规定内存分配失败将会导致一个 `bad_alloc` 异常。标准库以及第三方库，将会大量使用自由存储。由于内存分配请求会在最不期望它发生时发生，所以自己写的任何程序都应该为处理 `bad_alloc` 异常做好准备。显而易见的方法是提供一个处理程序：

```
catch(const bad_alloc& x)
{
    cerr<<"out of memory:"<<x.what()<<endl;
    abort();
}
```

如果更偏爱经典的返回空指针的行为，可以使用 `new` 运算符的特殊形式：

```
#include <new>
//...
T* tp=new (nothrow) T;
if (tp)
    //使用 tp...
```

而另一种处理内存不足情况的方法是取代其本身的部分内存分配机制。当内存分配失败时，C++ 就会调用缺省的 `new_handler`，它将按顺序抛出 `bad_alloc` 异常。可以通过把处理函数的地址传递到 `set_new_handler` 函数来建立自己的处理函数（参见程序清单 20.8）。

20.6 重载 `new` 和 `delete`

标准 C++ 库定义了 12 个用于分配和释放内存的函数。当使用 `new` 运算符时，它会计算出所需的字节数然后会按顺序调用这些函数中的一个来分配这些字节。这 12 个函数是：

```
//标量形式
void *operator new(size_t);
void *operator new(size_t, const nothrow_t);
void operator delete(void *);
void operator delete(void *, const nothrow_t);
```

```
//数组形式
void *operator new[](size_t);
void *operator new[](size_t, const nothrow_t);
void operator delete[](void *);
void operator delete[](void *, const nothrow_t);
```

```
//配置 new
void *operator new (size_t, void *);
void *operator new[](size_t, void *);
void operator delete(void *, void *);
void operator delete[](void *, void *);
```

无论什么时候，当从堆中给一个内置类型的对象分配空间时，`new` 就会调用上面的第一个 `operator new` 来获得内存。通过 `malloc` 可以很容易地实现它（见程序清单 20.9）。同样地，删除一个动态的内置类型对象将导致调用 `::operator delete (void *)`。程序清单 20.10 也说明了可以覆盖这些函数。注意，在替代的格式中，我使用 `printf` 而不是 `cout`。在许多 C++ 的实现中，`cout` 使用 `new` 运算符，因此，如果我在 `operator new` 中使用 `cout` 的话，我将造成无限的递归！

当用与上面函数中的前 8 个函数相同的形式定义任何的函数时，它将在程序中替换该函数的库函数版本。后面 4 个配置形式都属于重载，它们不像其他的函数那样可以被替代。由于许多标准和第三方便序组件依赖全局的 `new` 和 `delete`，所以替代它们根本不是一种好的办法。也可以在每一个类的基础上，替代运算符 `new` 和 `delete` 的各种形式（参见下面的“堆的管理”）。

程序清单 20.8 举例说明 `set_new_handler`

```
// exhaust1.cpp
#include <iostream>
#include <stdlib.h>
#include <new>
inline void my_handler()
{
    cout << "Memory exhausted" << endl;
    abort();
}
main()
{
    set_new_handler(my_handler);

    for (int i = 0; ; ++i)
    {
        (void) new double[100];
        if ((i+1)%10 == 0)
            cout << (i+1) << " allocations" << endl;
    }
}
// 输出:
10 allocations
20 allocations
```



```

30 allocations
40 allocations
50 allocations
60 allocations
70 allocations
Memory exhausted
Abnormal program termination

```

20.7 配置 new

有时需要在一个预定的地址上创建一个对象。这个地址的位置可能在某个设备映射的 RAM 区域中，或者在一个特殊类的堆中。可以为 new 运算符构建一个使用配置语法的对象。例如为了在地址 p 构建 T，可以像下面这样做：

```

#include<new>
T*tp=new (p) T;

```

程序清单 20.9 ::operator new 和:: operator delete 的典型实现

```

// opnew.cpp
#include <stdlib.h>
#include <new.h>

void *operator new(size_t siz)
{
    //获得 new_handler
    void (*new_handler)() = set_new_handler(0);
    set_new_handler(new_handler);

    for (;;)
    {
        //成功则返回指针
        void *p = malloc(siz);
        if (p)
            return p;

        //如果有处理函数，调用它
        if (new_handler)
            new_handler();
        else
            return 0;
    }
}

void operator delete(void *p)
{
    if (p)

```

```

        free(p);
    }

```

在这种情况下，由于内存的位置已经定了下来，因此没有任何形式的 `Operatornew` 执行。实际上，配置 `operator new` 的默认版本忽略了它的 `size` 参数，而只是返回其地址参数：

```

void * operator new(size_t, void *p)
{
    return p;
}

```

如程序清单 20.11 所示，可以用任意多的参数重载（而不是覆盖）配置 `new`。

可以使用配置 `new` 来使赋值运算符和它的拷贝构造函数同步，而不用复制初始化 `new` 的代码。可以像下面这样通过对象的 `this` 指针显式地调用析构函数，并在适当的位置用配置 `new` 来重建新的拷贝函数。如下所示：

```

T& T::operator=(const T& x)
{
    if (this != &x)
    {
        this->T::~~T();
        new (this) T(x);
    }
    return *this;
}

```

程序清单 20.10 覆盖 `operator new` 和 `operator delete`

```

// override.cpp
#include <stdio.h>
#include <stdlib.h>
#include <new>

void* operator new(size_t siz)
{
    printf("allocating %d bytes\n",siz);
    return malloc(siz);
}

void operator delete(void *p)
{
    printf("deleting memory at %p\n",p);
    free(p);
}

main()
{
    double *dp = new double;
    delete dp;
}

```

```

}

// 输出:
allocating 8 bytes
deleting memory at 007C2C9C

```

程序清单 20.11 一个有两个参数的配置 operator new

```

// overload.cpp
#include <iostream.h>

void *operator new(size_t siz, void *arg1, int arg2)
{
    cout << "new: siz == " << siz
          << ", arg1 == " << (void *) arg1
          << ", arg2 == " << arg2 << endl;
    return arg1;
}

main()
{
    void *p = (void *) 0x1234;
    int *ip = new (p,100) int;
    cout << "ip == " << (void *) ip << endl;
    return 0;
}

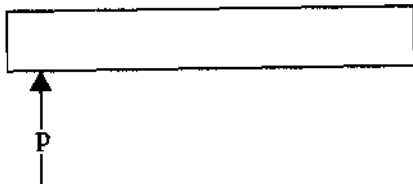
// 输出:
new: siz == 2, arg1 == 0x1234, arg2 == 100
ip == 0x1234

```

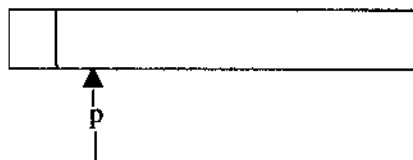
然而，这种很聪明的技巧也许会因为派生类而中止。想了解更多的信息，请参阅 Pete,Becker, *Not All operator==s Are Equal C/C++ Users Journal* (May 1997), pp.91-92。

20.8 堆的管理

你是否曾想过，当调用 `free(p)` 时，堆管理机制是如何知道有多少内存需要释放的呢？由于你并没有告诉堆管理机制这些信息，所以它一定在某些地方保存了这些信息。在大多数系统中，保存这些信息的地方就在 `p` 指针本身之前，因此并没得到了这个：

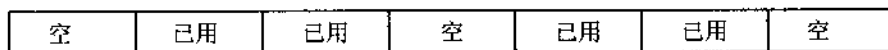


而是得到了这个：



当你分配了非常多的小的对象时，内存的额外开销是相当巨大的。

存在的另一个现象就是内存碎片。当分配了许多对象并删除一些后，在堆中就会出现间隙：



尽管在堆栈中有足够的空间来容纳另一个对象，但是空间却不连续，因此内存分配也许会失败。一种可以回避这些问题的办法是分配一个大的内存池，并且由自己来管理它。这样做的结果是，为容器类节省了时间和空间。

管理特殊类的堆时使用配置 `new` 是非常方便的。程序清单 20.12 中的程序有一个类模板用于可扩展的向量。为了节约额外开销，这个向量分配了一块存储区用于容纳预先确定的元素的数量（CHUNK）。每当把一个元素附加给这个向量时，它都只是在下一个可用的位置上构建这个元素：

```
new (arena + length_++) T(x);
```

当需要扩展基本的存储区时，只使用全局的 `operator new`。

```
T new_arena=(T*) ::operator new(sizeof(T)*new_capacity);
```

当需要销毁单个元素的时候，由于该块内存是存储区的一部分并且可以重用，所以需要重新分配任何内存。相反，可以显式地调用析构函数：

```
(arena+i)->T::~T( );
```

关于 `Vector` 类的使用，见程序清单 20.13。

20.9 避免内存管理

由于 C 和 C++ 很容易产生对动态内存的错误管理，所以它们受到了很多批评。一个解决这些问题的好方法是如果可能的话让别人为你做这些工作。例如，如果不想像程序清单 20.1 的分类程序中那样把行读进内存，那么何不让标准库做这个最困难的部分呢？程序清单 20.14 中的程序使用了 `getline` 函数来读取任意长度的行，然后把它们存到一个能处理自身内存的向量中。上面介绍过的 `arglist` 机制也能很容易地用于向量（参见第 16 章）。

程序清单 20.12 一个可管理自身堆的 `Vector` 模板类

```
// Vector.h
#include <iostream>
#include <stddef.h>
```

```

#include <stdlib.h>
#include <new>

// 下两个函数正好是为了追踪执行:
inline void *operator new(size_t siz)
{
    cout << ">>> operator new (" << siz << " bytes)" << endl;
    return malloc(siz);
}

inline void operator delete(void *p)
{
    cout << ">>> operator delete: " << (void *) p << endl;
    free(p);
}

template<class T>
class Vector
{
public:
    Vector();
    Vector(size_t);
    ~Vector();

    // 搜索路径和下标:
    Vector<T>& operator+=(const T&);
    T& operator[](size_t);

    // 与长度相关的函数:
    size_t length() const;
    void resize(size_t);
    size_t capacity() const;
    void reserve(size_t);

private:
    T *arena;    // 特殊类存储区域
    size_t length_;
    size_t capacity_;

    enum {CHUNK = 10};

    // 不允许复制和赋值:
    Vector(const Vector&);
    Vector<T>& operator=(const Vector<T>&);
    static size_t next_chunk(size_t n);
};

```

```
template<class T>
inline Vector<T>::Vector()
{
    // 初始化空的向量
    arena = 0;
    length_ = capacity_ = 0;
}
template<class T>
inline Vector<T>::Vector(size_t n)
{
    // 分派多重 CHUNK 元素>= n
    length_ = 0;
    capacity_ = next_chunk(n);
    arena = (T *) ::operator new(sizeof(T) * capacity_);
    cout << ">>> arena created at " << (void *) arena << endl;
}
template<class T>
Vector<T>::~~Vector()
{
    // 为每个元素执行析构函数
    for (int i = 0; i < length_; ++i)
        (arena+i)->T::~~T();

    ::operator delete(arena);
}
template<class T>
inline T& Vector<T>::operator[](size_t pos)
{
    if (pos >= length_)
        throw "bad index in Vector<T>::operator[]";
    return arena[pos];
}
template<class T>
inline size_t Vector<T>::length() const
{
    return length_;
}
template<class T>
inline size_t Vector<T>::capacity() const
{
    return capacity_;
}
template<class T>
void Vector<T>::reserve(size_t new_capacity)
{
    // 只允许增加:
```

```

    if (new_capacity > capacity_)
    {
        new_capacity = next_chunk(new_capacity);
        if (new_capacity > capacity_)
        {
            // 复制元素到新的空间
            T *new_arena = (T*) ::operator new(sizeof(T) *
                                                new_capacity);
            cout << ">>> new arena created at "
                  << (void *) new_arena << endl;
            for (int i = 0; i < length_; ++i)
                (void) new (new_arena + i) T(arena[i]);

            // 破坏旧的向量
            for (i = 0; i < length_; ++i)
                (arena+i)->T::~~T();
            delete arena;

            // 更新状态
            arena = new_arena;
            capacity_ = new_capacity;
        }
    }
}

template<class T>
void Vector<T>::resize(size_t new_length)
{
    // 只允许减少:
    if (new_length < length_)
    {
        // 只销毁截尾元素
        // 不要改变容量
        for (int i = new_length; i < length_; ++i)
            (arena+i)->T::~~T();
        length_ = new_length;
    }
}

template<class T>
Vector<T>& Vector<T>::operator+=(const T& x)
{
    if (length_ == capacity_)
        reserve(length_ + 1);
    (void) new (arena + length_++) T(x);
    return *this;
}

```

```
template<class T>
inline size_t Vector<T>::next_chunk(size_t n)
{
    return ((n + CHUNK - 1) / CHUNK) * CHUNK;
}
```

程序清单 20.13 测试 Vector 类

```
// tvector.cpp
#include <iostream.h>
#include "Vector.h"

// 用于测试向量的用户定义类
class Foo
{
    long x;

public:
    Foo() : x(0)
    {
        cout << "Foo::Foo()\n";
    }

    Foo(int i) : x(i)
    {}

    Foo(const Foo& f) : x(f.x)
    {
        cout << "Foo::Foo(const Foo&)\n";
    }

    ~Foo()
    {
        cout << "Foo::~~Foo()\n";
    }

    friend ostream& operator<<(ostream& os, const Foo& f)
    {
        cout << f.x; return os;
    }
};

main()
{
    // 实例化整型向量
    Vector<int> v(5);
    for (int i = 0; i < 11; ++i)
        v += i;
```



```

    for (i = 0; i < v.length(); ++i)
        cout << v[i] << endl;

    // 实例化 Foo 向量
    Vector<Foo> v2;
    v2 += 0;
    v2 += 1;
    v2 += 2;
    for (i = 0; i < v2.length(); ++i)
        cout << v2[i] << endl;

    return 0;
}

```

// 输出:

```

>>> operator new (20 bytes)
>>> arena created at 0x181e
>>> operator new (40 bytes)
>>> new arena created at 0x1836
>>> operator delete: 0x181e
0
1
2
3
4
5
6
7
8
9
10
>>> operator new (40 bytes)
>>> new arena created at 0x1862
Foo::Foo(const Foo&)
Foo::~~Foo()
Foo::Foo(const Foo&)
Foo::~~Foo()
Foo::Foo(const Foo&)
Foo::~~Foo()
0
1
2
Foo::~~Foo()
Foo::~~Foo()
Foo::~~Foo()
>>> operator delete: 0x1862

```

```
>>> operator delete: 0x1836
```

程序清单 20.14 程序清单 20.1 的更好的版本

```
// sort.cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

main(int argc, char *argv[])
{
    vector<string> strings;

    ifstream ifs(argv[1]);
    if (!ifs)
        return 1;

    string line;
    while (getline(ifs, line))
        strings.push_back(line);

    sort(strings.begin(), strings.end());

    for (int i = 0; i < strings.size(); ++i)
        cout << strings[i] << endl;

    return 0;
}
```

20.10 小结

- 当预先不知道需要多少内存空间来存储一个对象，或者不知道需要存储多少个对象时，可以使用动态内存。
- `new` 运算符调用 `operator new` 来为一个动态对象分配内存，然后再调用适当的构造函数。`delete` 运算符通过调用对象的析构函数来销毁这个对象，然后调用 `operator delete`。
- 对于数组一定要使用 `delete[]`。
- 无论什么时候，当一个类有指针数据成员时，都要检查是否需要使用深拷贝。
- 记住对内存操作的失败将引发 `bad_alloc` 异常。如果想要经典的“返回空指针”行为，可以使用 `new` 的 `nothrow` 版本。进一步可以通过提供一个 `new` 处理函数自定义内存恢复。

- 在给定的地址上使用配置 `new` 来构建一个对象。
- 为了使内存碎片的数量尽可能少，可以考虑将动态对象的同类集合分配到一个专用内存池中，如容器。
- 无论什么时候，只要有可能就让标准库来为你管理内存。

附录 A C/C++的兼容性

为了确保完成类型安全这个任务，同时也是为了支持面向对象程序设计的风格，C++必须服从C语言中的一些规则。下面总结了这两种语言之间的一些主要的不兼容性。即使你是一个C程序员，遵循C++这些规则有可能会编写出更安全的程序。

1. 新的注释风格。C++把//符号之后的所有文本都视为注释。

2. 新的关键字。C++在保留了C中许多关键字的同时也增加了大量新的关键字，并且不允许它们作为用户自定义的标识符使用。下面是扩展的关键字列表：

and	do	not	this
and_eq	double	not_eq	throw
asm	dynamic	operator	true
auto	else	or	try
bitand	enum	or_eq	typedef
bitor	explicit	private	typeid
bool	extern	protected	typename
break	false	public	union
case	float	reinterpret_cast	unsigned
catch	friend	return	using
char	goto	short	virtual
class	if	signed	void
compl	inline	sizeof	volatile
const	int	static	wchar_t
const_cast	long	static_cast	volatile
continue	mutable	struct	xor
default	namespace	switch	xor_eq
delete	new	template	

3. 新的标识符。一些新的标识符（参见第2和第3章）。

<:	%: %:
:>	::
<%	.*
%>	->*
%:	

4. 字符文字的类型。一个字符文字，如‘a’，在C++中是字符型的（相对于整型它允许重载）。而在C中，字符文字是整型的。

5. C风格字符串文字常量。一个文字如“A”是const char*类型。在C中，这并不是一个

个 `const`。这意味着不赞成使用如下这种表达式：

```
char* p = "abc";    //把 const 赋值给非 const
```

6. 常量的特性。在 C++ 中，所有的整型常量都是真正的编译期常量，并且它们还可以作为数组维数使用。所有在文件作用域内声明的常量对象都有内部的链接，并且允许用它们替换头文件中类对象的宏（参见第 11 章）。

7. `main()` 的特性。在 C++ 中 `main()` 函数不能被递归地调用。

8. 安全类型连接。有不同标记的函数不能被错误地连接（参见第 1 章）。

9. `void*` 的特性。把指向 `void` 的指针赋值给其他任何类型的指针时需要强制类型转换（参见第 10 章）。

10. 结构的标记是类型名。现在，结构的标记与类型定义和正常的变量名具有相同的名字空间。这就允许在定义用户对象时可以不使用 `struct` 和 `class` 关键字。

11. 没有隐式的整型。整型不再假定为是函数返回值的类型。必须显式地指定函数返回值的类型。

12. 枚举类型与整型的兼容性。将整型赋值给枚举类型时需要强制类型转换。

13. 枚举类型可重载。现在，枚举类型在函数重载时被看作是特殊的类型。

14. 要求有函数原型。在旧式 C 风格的函数声明中，不允许在参数列表外声明参数类型。所有的函数在被调用之前必须被完全地声明或定义。

15. 空参数列表 == `void`。一个没有参数的函数——`f()`，就相当于 `f(void)`。

16. 结构定义了一个作用域。嵌套的结构体定义属于它们被定义地方的作用域。

附录 B 标准 C++ 算法

下表中的模板参数使用了如下的简洁表示：

缩写	意义
II	输入迭代器 (input iterator)
OI	输出迭代器 (output iterator)
FI	前向迭代器 (forward iterator)
BI	双向迭代器 (bi-directional iterator)
RI	随机访问迭代器 (random access iterator)
UOP	一元操作符 (unary operator)
BOP	二元操作符 (binary operator)
P	谓词 (predicate)
BinP	二元谓词 (binary predicate)
Gen	发生器 (generator)
RandGen	随机数发生器 (random number generator)

我用一些复杂的符号来做适当的表示。例如，使用表达式 $O(3*N)$ 看起来似乎有点笨拙，但是这样做是为了表明操作数不超出 $3*N$ 。复杂的方式并没有用“BIG-OH”符号来表示一个精确的操作数。

表 B.1

非变异的序列算法

函 数	复 杂 度
<code>template<class II, class P> F for_each (II first, II last, F f) ;</code>	N
<code>template<class II, class T></code> <code>II find (II first, II last, const T& value) ;</code>	$O(N)$
<code>template<class II, class P></code> <code>II find_if (II first, II last, P pred) ;</code>	$O(N)$
<code>template<class FI1, class FI2></code> <code>FI1 find_end (FI1 first1, FI1 last1, FI2 first2, FI2 last2) ;</code>	$O(M * (N - M + 1))$

续表

函 数	复 杂 度
<pre>template<class FI1, class FI2, class BinP> FI1 find_end (FI1 first1, FI1 last1, FI2 first2, FI2 last2, BinP pred);</pre>	$O(M * (N - M + 1))$
<pre>template<class FI1, class FI2> FI1 find_first_of (FI1 first1, FI1 last1, FI2 first2, FI2 last2);</pre>	$O(M * N)$
<pre>template<class FI1, class FI2, class BinP> FI1 find_first_of (FI1 first1, FI1 last1, FI2 first2, FI2 last2, BinP pred);</pre>	$O(M * N)$
<pre>template<class FI> FI adjacent_find (FI first, FI last);</pre>	$O(N)$
<pre>template<class FI, class BinP> FI adjacent_find (FI first, FI last, BinP pred);</pre>	$O(N)$
<pre>template<class II, class T> iterator_traits<II>::difference_type count (II first, II last, const T& value);</pre>	N
<pre>template<class II, class P> iterator_traits<II>::difference_type count_if (II first, II last, P pred);</pre>	N
<pre>template<class II1, class II2> pair<II1, II2> mismatch (II1 first1, II1 last1, II2 first2);</pre>	$O(N)$
<pre>template<class II1, class II2, class BinP> pair<II1, II2> mismatch (II1 first1, II1 last1, II2 first2, BinP pred);</pre>	$O(N)$
<pre>template<class II1, class II2> bool equal (II1 first1, II1 last1, II2 first2);</pre>	$O(N)$
<pre>template<class II1, class II2, class BinP> bool equal (II1 first1, II1 last1, II2 first2, BinP pred);</pre>	$O(N)$
<pre>template<class FI1, class FI2> FI1 search (FI1 first1, FI1 last1, FI2 first2, FI2 last2);</pre>	$O(M * N)$
<pre>template<class FI1, class FI2, class BinP> FI1 search (FI1 first1, FI1 last1, FI2 first2, FI2 last2, BinP pred);</pre>	$O(M * N)$

续表

函 数	复 杂 度
<pre>template<class FI1, class Size, class T> FI search_n (FI first, FI last, Size count, const T& value);</pre>	N*count
<pre>template<class FI1, class Size, class T, class BinP> FI search_n (FI first, FI last, Size count, const T& value , BinP pred);</pre>	N*count

表 B.2

变异的序列算法

函 数	复 杂 度
<pre>template<class II, class OI> OI copy (II first, II last, OI result);</pre>	N
<pre>template<class BI1, class BI2> BI2 copy_backward (BI1 first, BI1 last, BI2 result);</pre>	N
<pre>template<class T> void swap (T& a, T& b);</pre>	常数
<pre>template<class FI1, class FI2> FI2 swap_ranges (FI1 first1, FI1 last1, FI2 first2);</pre>	N
<pre>template<class FI1, class FI2> void iter_swap (FI1 a, FI2 b);</pre>	常数
<pre>template<class II, class OI, class UOP> OI transform (II first, II last, OI result, UOP op);</pre>	N
<pre>template<class II1, class II2, class OI, class BOP> OI transform (II1 first1, II1 last1, II2 first2, OI result, BOP op);</pre>	
<pre>template<class FI, class T> void replace (FI first, FI last, const T& old_value, const T& new_value);</pre>	N
<pre>template<class FI, class P, class T> void replace_if (FI first, FI last, P pred, const T& new_value);</pre>	N
<pre>template<class II, class OI, class T> OI replace_copy (II first, II last, OI result, const T& old_value, const T& new_value);</pre>	N
<pre>template<class II, class OI, class P, class T> OI replace_copy_if (II first, II last, OI result, P pred, const T& new_value);</pre>	N

续表

函 数	复 杂 度
template<class FI, class T> void fill (FI first, FI last, const T& value) ;	N
template<class OI, class Size, class T> void fill_n (OI first, Size n, const T& value) ;	N
template<class FI, class Gen> void generate (FI first, FI last, Gen gen) ;	N
template<class OI, class Size, class Gen > void generate_n (OI first, Size n, Gen gen) ;	N
template<class FI, class T> FI remove (FI first, FI last, const T& value) ;	N
template<class FI, class P> FI remove_if (FI first, FI last, P pred) ;	N
template<class II, class OI, class T> OI remove_copy (II first, II last, OI result, const T& value) ;	N
template<class II, class OI, class P> OI remove_copy_if (II first, II last, OI result, P pred) ;	N
template<class FI> FI unique (FI first, FI last) ;	N-1
template<class FI, class BinP> FI unique (FI first, FI last, BinP pred) ;	N-1
template<class II, class OI> OI unique_copy (II first, II last, OI result) ;	N
template<class II, class OI, class BinP > OI unique_copy (II first, II last, OI result, BinP pred) ;	N
template<class BI> void reverse (BI first, BI last) ;	N/2
template<class BI, class OI> OI reverse_copy (BI first, BI last, OI result) ;	N
template<class FI> void rotate (FI first, FI middle, FI last) ;	O (N)
template<class FI, class OI> OI rotate_copy (FI first, FI middle, FI last, OI result) ;	N

续表

函 数	复 杂 度
<code>template<class RI> void random_shuffle (RI first, RI last) ;</code>	$N-1$
<code>template<class RI, class RandGen></code> <code>void random_shuffle (RI first, RI last, RandGen& rand) ;</code>	$N-1$
<code>template<class BI, class P></code> <code>BI partition (BI first, BI last, P pred) ;</code>	N ($N/2$ swaps)
<code>template<class BI, class P></code> <code>BI stable_partition (BI first, BI last, P pred) ;</code>	N Swaps: $O(N\log N)$

表 B.3

排序算法

函 数	复 杂 度
<code>template<class RI>void sort (RI first, RI last) ;</code>	$O(N\log N)$
<code>template<class RI, class Compare></code> <code>void sort (RI first, RI last, Compare comp) ;</code>	$O(N\log N)$
<code>template<class RI>void stable_sort (RI first, RI last) ;</code>	$O(N(\log N)^2)$
<code>template<class RI, class Compare></code> <code>void stable_sort (RI first, RI last, Compare comp) ;</code>	$O(N(\log N)^2)$
<code>template<class RI></code> <code>void partial_sort (RI first, RI middle, RI last) ;</code>	$O(N\log N)$
<code>template<class RI, class Compare ></code> <code>void partial_sort (RI first, RI middle, RI last, Compare comp) ;</code>	$O(N\log N)$
<code>template<class II, class RI></code> <code>RI partial_sort_copy (II first, II last, RI result_first, RI result_last) ;</code>	$O(N\log N)$
<code>template<class II, class RI, class Compare ></code> <code>RI partial_sort_copy (II first, II last, RI result_first, RI result_last, Compare comp) ;</code>	$O(N\log N)$
<code>template<class RI> void nth_element (RI first, RI nth, RI last) ;</code>	$O(N)$
<code>template<class II, class RI, class Compare ></code> <code>void nth_element (RI first, RI nth, RI last, Compare comp) ;</code>	$O(N)$

表 B.4

排序查找和合并算法

函 数	复 杂 度
template<class FI, class T> FI lower_bound (FI first, FI last, const T& value) ;	$O(\log N)$
template<class FI, class T, class Compare> FI lower_bound (FI first, FI last, const T& value, Compare comp) ;	$O(\log N)$
template<class FI, class T> FI upper_bound (FI first, FI last, const T& value) ;	$O(\log N)$
template<class FI, class T, class Compare> FI upper_bound (FI first, FI last, const T& value, Compare comp) ;	$O(\log N)$
template<class FI, class T> pair<FI, FI> equal_range (FI first, FI last, const T& value) ;	$O(\log N)$
template<class FI, class T, class Compare> pair<FI, FI> equal_range (FI first, FI last, const T& value, Compare comp) ;	$O(\log N)$
template<class FI, class T> bool binary_search (FI first, FI last, const T& value) ;	$O(\log N)$
template<class FI, class T, class Compare > bool binary_search (FI first, FI last, const T& value, Compare comp) ;	$O(\log N)$
Template<class I1, class I2, class OI> OI merge (I1 first1, I1 last1, I2 first2, I2 last2, OI result) ;	$O(M*N)$
template<class I1, class I2, class OI, class Compare> OI merge (I1 first1, I1 last1, I2 first2, I2 last2, OI result, Compare comp) ;	$O(M*N)$
template<class BI> void inplace_merge (BI first, BI mid, BI last) ;	$O(N \log N)$
template<class BI, class Compare> void inplace_merge (BI first, BI mid, BI last, Compare comp) ;	$O(N \log N)$

表 B.5

集合操作

函 数	复 杂 度
template<class I1, class I2> bool includes (I1 first1, I1 last1, I2 first2, I2 last2) ;	$O(M+N)$

续表

函 数	复 杂 度
template<class II1, class II2, class Compare > bool includes (II1 first1, II1 last1, II2 first2, II2 last2, Compare comp) ;	$O(M+N)$
template<class II1, class II2, class OI> OI set_union (II1 first1, II1 last1, II2 first2, II2 last2, OI result) ;	$O(M+N)$
template<class II1, class II2, class OI, class Compare > OI set_union (II1 first1, II1 last1, II2 first2, II2 last2, OI result, Compare comp) ;	$O(M+N)$
template<class II1, class II2, class OI> OI set_intersection (II1 first1, II1 last1, II2 first2, II2 last2, OI result) ;	$O(M+N)$
template<class II1, class II2, class OI, class Compare > OI set_intersection (II1 first1, II1 last1, II2 first2, II2 last2, OI result, Compare comp) ;	$O(M+N)$
template<class II1, class II2, class OI> OI set_difference (II1 first1, II1 last1, II2 first2, II2 last2, OI result) ;	$O(M+N)$
template<class II1, class II2, class OI, class Compare > OI set_difference (II1 first1, II1 last1, II2 first2, II2 last2, OI result, Compare comp) ;	$O(M+N)$
template<class II1, class II2, class OI> OI set_symmetric_difference (II1 first1, II1 last1, II2 first2, II2 last2, OI result) ;	$O(M+N)$
template<class II1, class II2, class OI, class Compare > OI set_symmetric_difference (II1 first1, II1 last1, II2 first2, II2 last2, OI result, Compare comp) ;	$O(M+N)$

表 B.6

堆操作

函 数	复 杂 度
template<class RI> void push_heap (RI first, RI last) ;	$O(\log N)$
template<class RI, class Compare> void push_heap (RI first, RI last, Compare comp) ;	$O(\log N)$
template<class RI> void pop_heap (RI first, RI last) ;	$O(\log N)$

续表

函 数	复 杂 度
template<class RI, class Compare> void pop_heap (RI first, RI last, Compare comp) ;	$O(\log N)$
template<class RI> void make_heap (RI first, RI last) ;	$O(3*N)$
template<class RI, class Compare> void make_heap (RI first, RI last, Compare comp) ;	$O(3*N)$
template<class RI> void sort_heap (RI first, RI last) ;	$O(N \log N)$
template<class RI, class Compare> void sort_heap (RI first, RI last, Compare comp) ;	$O(N \log N)$

表 B.7

最小/最大值算法

函 数	复 杂 度
template<class T> const T& min (const T&a, const T&b) ;	常数
template<class T, class Compare> const T& min (const T& a, const T& b, Compare comp) ;	常数
template<class T> const T& max (const T& a, const T& b) ;	常数
template<class T, class Compare> const T& max (const T& a, const T& b, Compare comp) ;	常数
template<class FI> FI min_element (FI first, FI last) ;	$N-1$
template<class FI, class Compare > FI min_element (FI first, FI last, Compare comp) ;	$N-1$
template<class FI> FI max_element (FI first, FI last) ;	$N-1$
template<class FI, class Compare > FI max_element (FI first, FI last, Compare comp) ;	$N-1$
template<class II1, class II2> bool lexicographical_compare (II1 first1, II1 last1, II2 first2, II2 last2) ;	$\min(N, M)$
template<class II1, class II2, Compare comp > bool lexicographical_compare (II1 first1, II1 last1, II2 first2, II2 last2, Compare comp) ;	$\min(N, M)$

表 B.8

排列算法

函 数	复 杂 度
template<class BI> bool next_permutation (BI first, BI last) ;	O(N/2)
template<class BI, class Compare> bool next_permutation (BI first, BI last, Compare comp) ;	O(N/2)
template<class BI> bool prev_permutation (BI first, BI last) ;	O(N/2)
template<class BI, class Compare> bool prev_permutation (BI first, BI last, Compare comp) ;	O(N/2)

表 B.9

数值算法

函 数	复 杂 度
template<class II, class T> T accumulate (II first, II last, T init) ;	O(N)
template<class II, class T, class BOP> T accumulate (II first, II last, T init, BOP op) ;	O(N)
template<class II1, class II2, class T> T inner_product (II1 first1, II1 last1, II2 first2, T init) ;	O(N)
template<class II1, class II2, class T, class BOP1, class BOP2> T inner_product (II1 first1, II1 last1, II2 first2, T init, BOP1 op1, BOP2 op2) ;	O(N)
template<class II, class OI> OI partial_sum (II first, II last, OI result) ;	N-1
template<class II, class OI, class BOP> OI partial_sum (II first, II last, OI result, BOP op) ;	N-1
template<class II, class OI> OI adjacent_difference (II first, II last, OI result) ;	N-1
template<class II, class OI, class BOP> OI adjacent_difference (II first, II last, OI result, BOP op) ;	N-1

附录 C 函数对象和适配器

函数对象

所有的标准函数对象都是可改变的，因为它们定义了函数对象适配器所要求的参数和返回值类型。它们通过从 `unary_function` 或者从 `binary_fution` 派生来完成这项工作。关于更多的内容参见第 15 章。

表 C.1

算术运算

操作	操作元基数	operator () 返回值
加	二元	$x+y$
减	二元	$x-y$
乘	二元	$x*y$
除	二元	x/y
求模	二元	$x\%y$
取反	一元	$-x$

表 C.2

比较

操作	操作元基数	operator () 返回值
等于	二元	$x==y$
不等于	二元	$x!=y$
大于	二元	$x>y$
小于	二元	$x<y$
大于等于	二元	$x>=y$
小于等于	二元	$x<=y$

表 C.3

逻辑运算

操作	操作元基数	operator () 返回值
与	二元	$x \& y$
或	二元	$x y$
非	一元	$!x$

表 C.4

否定关系

对象	操作元基数	operator () 返回值
<code>unary_negate (pred)</code>	一元	$!pred (x)$
<code>binary_negate (pred)</code>	二元	$!pred (x,y)$

表 C.5

联编程序

对象	操作元基数	Operator (x) 返回值
<code>binder1st (op,value)</code>	一元	$op (value, x)$
<code>binder2nd (op,value)</code>	一元	$op (x, value)$

表 C.6

函数指针对象

对象	操作元基数	operator (x...) 返回值
<code>pointer_to_unary_function (f)</code>	一元	$f (x)$
<code>pointer_to_unary_function (f)</code>	二元	$f (x,y)$

表 C.7

成员函数指针对象

对象	操作元基数	operator (...) 返回值
<code>mem_fun_t (pmf)</code>	一元	$ptr2obj \rightarrow pmf ()$
<code>mem_fun1_t (pmf)</code>	二元	$ptr2obj \rightarrow pmf (x)$
<code>mem_ref_t (pmf)</code>	一元	$\&obj \rightarrow pmf ()$
<code>mem_ref1_t (pmf)</code>	二元	$\&obj \rightarrow pmf (x)$

函数对象适配器

函数对象适配器是函数模板，这些模板把一个可改变的函数对象连同可选参数转化成其

他可改变的函数对象。

表 C.8

否定关系

适配器	返回值
not1 (unary_pred)	unary_negate (unary_pred)
not2 (binary_pred)	binary_negate (binary_pred)

表 C.9

联编程序

适配器	返回值
bind1st (bop, val)	bind1st (bop, val)
bind2nd (bop, val)	bind2nd (bop, val)

表 C.10

函数指针

适配器	返回值
prt_fun (R (*pf) (A))	pointer_to_unary_function (pf)
prt_fun (R (*pf) (A1,A2))	pointer_to_unary_function (pf)

表 C.11

成员函数指针

适配器	返回值
mem_fun (R (T::*pmf) ())	mem_fun_t (pmf)
mem_fun (R (T::*pmf) (A))	mem_fun1_t (pmf)
mem_ref (R (T::pmf) ())	mem_ref_t (pmf)
mem_ref (R (T::pmf) (A))	mem_ref1_t (pmf)

附录 D 有注解的参考书目

就我看来, 下面的这些书是精华中的精华。这些毫无疑问都是有价值的著作。更完整的书目列表, 参见 C/C++ 用户杂志网站 <http://www.cuj.com>。每个条目都带有下面列表中描述性的限定词注释:

限定词	意义
INTRO	介绍性的
INTMD	中等水平的
ADV	高级的
REF	参考资料
DESIGN	面向对象设计
IDIOMS	包含习惯用语/模式

C 实践者书目

FEUER, ALAN R., *The C Puzzle Book*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall ESM, 1989. ISBN 0-13-109926-4. 本书就像 C 程序员按惯例必须经过的通道。尽管有些深奥和抽象, 作者还是严格地检测了你对重要概念的掌握。[INTMD, REF]

KERNIGHAN, BRIAN W., and RITCHIE, DENNIS M., *The C Programming Language*, 2nd ed. Englewood Cliffs, NJ: Prentice Hall, 1988. ISBN 0-13-110362-8. 精髓参考书, C 族民必备的学习手册。[INTRO, INTMD, ADV, REF]

KOENIG, ANDREW, *C Traps and Pitfalls*, Addison-Wesley, 1989, ISBN 0-201-17928-8. [INTRO, INTMD, ADV, REF] Koenig 是一位独一无二的专家, 非常善于交流。他阐明了一些“暗礁”, 抓住了许多不为人注意的要点。精炼, 可读性强, 不可或缺。本书中文版已由人民邮电出版社出版, 书名《陷阱与缺陷》

PLAUGER, P. J., *The Standard C Library*. Englewood Cliffs, NJ: Prentice-Hall, 1991. ISBN 0-13-131509-9. 不可企及。本书提供了掌握 C 语言所需的磨炼。[INTMD, ADV, REF]

PLUM, THOMAS, *Learning to Program in C*, 2nd ed. Plum Hall, 1989. ISBN 0-911537-08-2. [INTRO]

——, *C Programming Guidelines*, 2nd ed. Plum Hall, 1989, ISBN 0-911537-07-4. [INTRO, INTMD, REF, DESIGN]

PLUM, THOMAS, and BRODIE, JIM, *Efficient C*, Plum Hall, 1985. ISBN 0-911537-05-8.

极少人像 Tom 那样了解 C，他的书还把 C 专家级的深刻理解和全面的软体工程原则有机结合。他的书现在略显过时(主要是因为写在 ANSI C 以前)，但那些原则经得起时间考验。

[INTMD, ADV, DESIGN]

C++ 实践者书目

CARGILL, TOM, *C++ Programming Style*. Addison-Wesley, 1992. ISBN 0-201-56365-7. 就像从商有技巧一样，本书教你恰当地使用 C++ 类和对象的技术。(*C/C++ Users Journal*, April 1993 的书评). [INTMD, DESIGN, IDIOMS]

CLINE, MARSHALL, and LOMOW, GREG, *C++ FAQs*. Addison-Wesley, 1995. ISBN 0-201-58958-3. 日常工作实践中最有用的文本。对优良的牢固的工程实践的涵盖，达到了惊人的深度。傻瓜才会错过这本书。[INTMD, ADV, REF, DESIGN, IDIOMS]

COPLIEN, JAMES, *Advanced C++*. Addison-Wesley, 1992. ISBN 0-201-54855-0. 唯一真正高级的书籍。领先于它的时代。富含设计术语。适合有实力的读者。
[ADV, DESIGN, IDIOMS]

ECKEL, BRUCE, *Thinking in C++*. Upper Saddle River, NJ: Prentice-Hall, 1995. ISBN 0-13-9177094. 目前为止最适合学习 C++ 的方法。一部美妙的教学著作。[INTRO]

HENRICSON, MATS, and NYQUIST, ERIK, *Industrial Strength C++*. Upper Saddle River, NJ: Prentice-Hall, 1997. ISBN 0-13-120965-5. 以精炼，可读的风格教授现实世界中的 C++ 编程 [REF, DESIGN, IDIOMS]

KOENIG, ANDREW, and MOO, BARBARA, *Ruminations on C++*. Addison-Wesley, 1997. ISBN 0-201-42339-1. 对我而言，这是我拥有的最好的 C++ 书籍。(*C/C++ Users Journal* 书评, April 1997). 本书中文版《C++ 沉思录》已由人民邮电出版社出版。
[INTMD, ADV, DESIGN, IDIOMS]

LIPPMAN, STANLEY, *C++ Primer*. 2nd ed. Addison-Wesley, 1991. ISBN 0-201-16487-6. 优秀的初学者教材，非常全面。(3rd Edition to appear). [INTRO]

MEYERS, SCOTT, *Effective C++*. 2nd ed. Addison-Wesley, 1998. ISBN 0-201-92488-9.

——, *More Effective C++*. Addison-Wesley, 1996. ISBN 0-201-63371-X.

目前为止，最好的两本书。精炼的章节，讲解特定的实现问题

[INTMD, REF, DESIGN, IDIOMS]

STROUSTRUP, BJARNE, *The C++ Programming Language*. 3rd ed. Addison-Wesley, 1997. ISBN 0-201-88954-4. 是“K&R”的 C++ 版本，但更易信。对 C++ 的覆盖最全面。
[INTRO, INTMD, REF]

——, *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0-201-54330-2. The standard text on C++ history and philosophy from its designer. Explains why C++ is the way it is. A must-read. [INTMD, ADV, IDIOMS]

来自设计者的关于 C++ 历史和哲学的标准文本。解释了 C++ 为何如其所现。必读之作。

附录 E C++标准的制定

(采访 Bjarne Stroustrup)

随着本书的出版,第二届官方委员会草案(CD2)已经发布,美国的各家制定标准的团体对 ISO/ANSI 标准联盟委员会(X3J16/wG21)都发表了各自的看法。C++语言的特点已经稳定一段时间了。就在 1996 年 7 月委员会在瑞典的斯德哥尔摩会议上通过 CD2 草案之前不久,我有幸以 *C/C++ Users Journal* 杂志的名义采访了 Bjarne Stroustrup,他对 C++语言的现状和未来发展前景的见解都很令我感兴趣。这部分是采访的摘录。

为了表示对制定标准所做的工作和接下来的采访的重视,我们首先回顾一下 C++的历史。有关 C++详细的、技术的、有趣的历史,请参阅 Stroustrup 的 *The Design and Evolution of C++* (Addison-Wesley 出版,1994 年)。

Bjarne Stroustrup 是丹麦人,在英国的剑桥大学获得博士学位,在研究工作中曾经将 Simula 语言用于分布式系统的仿真。由于他对该语言糟糕的性能很失望,所以在 1979 年当他的新雇主(AT&T Bell 实验室)邀请他加入去做“一些有趣的事情”时,他决定将 C 语言和他非常熟悉的 Simula 最显著的特征——类加以结合。这样,带类的 C 诞生了,它流行于 AT&T,并被称为“C++”,然后,这种语言开始成为它的发明者技术支持的负担。Stroustrup 的 *The C++ Programming Language* 第一版(Addison-Wesley,1985 年)出版后销售一空,这种语言变得十分流行。开始正如你所想象的那样,出现了多种 C++实现平台,每种都有各自的特色。大约是 C 语言标准成为正式标准的时候,C++委员会主要成员正在推进 C++标准。ANSI 委员会 X3J16 第一次会议于 1989 年 12 月召开,惠普公司的 Dmitry Lenkov 担任主席。1996 年 Sun Microsystems 公司的 Steve Clamage (C++早期的倡导者)成为主席。

委员会的基础文件包括 ISO 的 C 语言标准以及 Ellis 和 Stroustrup 的 *The Annotated C++ Reference Manual* (ARM),后者更多地反映了 AT&T C++ 2.0 的特点及 Bjarne 的扩充思想(主要包括模板和异常),开始的主要目标是建立标准化 IO 流,并且把模板和异常加到语言中。由于有大量的非美国成员,委员会投票表决从 1991 年 6 月在瑞典 Lund 召开的会议开始与 ISO 第 21 工作组联合。直到 1993 年末,事情好像进行得很顺利,然而,由于标准库中缺乏鲁棒性,依然存在着一一些令人不安的因素。在 1993 年 11 月召开的 San Jose 会议上,Alex Stepanov 做了一个关于真正应用模板的泛型编程介绍。到了第二年 3 月的 San Diego 会议上,他已经将他和 Meng Lee 的标准模板库(STL)精练到了委员会准备认真考虑的地步,即使这意味着将延迟标准的完成。记得当时我是一名保守的怀疑者,但还是举手赞同了 STL,而 STL 和 IO 流现在是标准 C++库的核心部分。

从 C++ 2.0 开始增加的主要特征包括模板、异常、用户名字空间和运行期类型识别 (RTTI)，它们之所以“主要”是因为它们直接影响着程序的所有结构；而次要特征只是缺乏插入性，它们本身的作用仍是十分强大的，包括新风格的类型转换、一种新型的布尔数据类型 (bool)、对枚举类型的重载能力、支持长字符，并且以变换标记来支持国外的键盘（例如 or 对应 II）。除了 STL 之外，标准库还包括改进过的流类、带有专门用于长字符和短字符的字符串类模板、用于 RTTI 的低层结构和重载及重写运算符 new 和 delete。

Chuck Allison: 我知道你有应用数学的博士学位，你还拥有其他哪些学位？

Bjarne Stroustrup: 情况并非完全如此，我在丹麦的奥尔胡斯大学拿到的是“数学及计算机科学”硕士学位 (Cand.Scient)，我之所以学习数学部分是因为在当时，那是去做计算机科学研究的唯一的途径。我在英国剑桥大学拿到的是“计算机科学”博士学位。我虽然不是一名好的数学家，但是我想总比不是一位数学家要强。

CA: 你是如何加入计算机专业的？

BS: 通过在大学里从事有关数学和计算机科学的工作。我已经尽力去回忆为什么我要做那个工作，但是我确实不知道，当时签约的时候甚至还没有看见过计算机，我想是理论和实践的结合吸引了我。

CA: 你什么时候注意到带类的 C 语言正在成为占据你大量时间的事情，如果它不是你的事业，也不是你的一个暂时的兴趣？

BS: 1982 年，我意识到对带类的 C 用户社区的技术支持正在成为我难以承受的负担，正如我看到的，用户数量太大以至于一个人难以很好地服务，而且我同时还在做研究工作。但是，人数太少又难以形成需要支持的基础组织。我不得不决定是否将带类的 C 发展成一种更灵活和更易于表达的语言，或者干脆放弃。幸运的是，我没有考虑第三种选择——人们提出的一种常规的解决方案——通过大肆宣传来召集更多的用户。

通过带类的 C，我决定把从 C 和我已经从 Simula 中获取的有价值的概念进一步结合起来，其结果就是 C++。以后有几次机会，我试图逃脱到其它的工作中，但是 C++ 团体的不断增长和新应用的挑战使我不得不继续专注其中。

CA: 你现在日常做什么工作？

BS: 我正在努力建立一个致力于大规模编程的研究团体，即研究大规模程序中编程的应用，而不仅仅是语言设计和小程序的研究，或者只专注于设计和处理。我认为编程技巧、编程语言和个别的编程者在大系统的开发中处于中心角色。非常普遍的现象是工业项目的规模或编程的角色被忽略了。

这种研究将涉及库和工具方面的工作。自然地，C++ 将在研究中扮演重要的角色，但是我并不打算将我的兴趣仅局限于 C++ 中。

我的部门是 AT&T 研究机构的一部分，AT&T 解体时，AT&T 保持了原 AT&T Bell 实验室信息科学研究机构的一部分，（另一部分成为朗讯 (Lucent Technologies)），现在 AT&T 部

分叫作 AT&T 研究部，我们的目标是成为世界第一的研究组织。

从很大程度上讲，C++依然是我的“日常工作”，我做标准研究、帮助用户、写文章、演讲及著书等等。偶尔我甚至也编写一些代码，尽管没有我想写的那么多。

CA: 在你退休前还想完成什么？

BS: 你指的是从 C++ 开发方面退休还是真正的退休？因为我离退休的年龄还很远，所以，我假设你指的是我在 C++ 方面的退休，我认为这种语言和它的标准库是完整的和健全的，这样 C++ 发展的阶段已经完成。ISO C++ 比任何早期的版本更能接近我的理想，对模板的改进和在标准库中包含 STL 是使 C++ 达到我的理想目标的关键。

CA: 你是怎样总结 STL 的影响或优点呢？

BS: 我最大的希望就是 STL 将教会更多的人把 C++ 作为一种高级的语言来用，而不是作为一种美化的汇编语言。我的 C++ 目标过去是、现在还是提高代码的抽象程度，这样，程序文本在任何合理的地方都可以直接反应应用的概念和一般的计算机科学概念。然而，太多的人迷失在费解的位和指针中。标准库允许人们按照库中所给的那样使用字符串、向量、表和图，缓解人们对 C 风格字符串以及基本数据结构指针形式实现的担心。

CA: 你经常用 Vasa 号的故事来鼓励用简单路线定义 C++ 语言，但是相当多的人都认为 C++ 是现有的最复杂的语言之一，你对这种观点有什么评述？如有不同层次的 C++ 用户，你怎样来辨别他们？

BS: 很明显这有一定的风险，为什么我还要费力地讲述一个警告性的故事呢？

在国王 Gustav 的要求下，Vasa 号的建造始于 1625 年，起初打算建造一艘常规的战船，但是在建造期间，国王在别的地方看见了更大更好的战船，因此，他改变了主意。他坚持要求把船建造成有两个炮台的旗舰，同时还坚持要求船上要有许多雕像以适合皇家旗舰的气派，结果这条船变得十分壮观，尽管它也相当重。在处女航中，Vasa 号只驶过了斯德哥尔摩港的一半就被一阵强风吹翻了，50 人死于这一事故。之后这艘船被打捞上来，你可以在斯德哥尔摩的博物馆中看到它。看上去它是那么美，比不扩充的第一次设计要美得多得多，即使它遭遇了 17 世纪战舰的通常命运，今天它仍然无与伦比的美，但这对它的设计者、建造者以及使用者并不是安慰。

迄今为止，C++ 已经逃脱了 Vasa 号的命运，尽管有很多充满希望的见解和可悲的预言，C++ 没有倾覆和消失。一部分被视为泛型的较新的扩展真正使 C++ 程序员的生活变得更加容易。通过数年的努力，我的程序已经明显地变得更短、更清晰、更容易编写。简单地说，我不再像以前那样经历许多弯路来表达一个好的设计思想。自然地，如果你认为最好的编程语言就是能够让新手通过一个示例就能进步神速的话，则 C++ 不是这样的，因为它的确需要较长时间的练习才能掌握。然而，我主要关心的是它所产生的代码，对于有能力的程序员，C++ 是令人愉快的工具。ISO C++ 比在 ARM 中描述的 C++ 更接近我的想法。

应值得注意的是 Gustav 国王在一个方面的考虑是对的：把 Vasa 号改为有两个炮台的“现

代”战舰、错误并不是增加一个炮台，炮台是 Vasa 号圆满完成使命的基本组成部分，错误的原因是增加炮台的方式。从这点上讲，我们应该用更友善的眼光来看待委员会花费额外的时间来确保对 C++ 的扩展，这是经过适当设计和严密实验的。

与原始的 pascal 语言相比，C++ 当然是相当复杂的，对于每一种其它的现代程序设计语言也是如此，然而，如果与我们所用的编程环境以及程序运行的系统相比，C++ 还是简单的。

对于初学者，试图先学习全部的 C++ 内容然后开始使用它，可能是一个非常严重的错误。最好从一种子集开始学习，而后再根据需要扩充自己的指令集，我推荐以类似 C 的子集开始，但是要使用标准向量、表、图和字符串类来取代令人迷惑的 C 风格的数组和字符串。自然地，应该避免使用宏，联合和位域也应避免使用。因此，应用 C 和 C++ 子集加上我刚提到的 4 个标准库，可以写出很好的代码。

而后，初学者便可以开始试着用简单的类、简单的模板和简单类的继承来编程了。我建议在设计任何重要类继承之前要把注意力放在抽象类上，即使很简单的异常处理，如果及早采用也是有益的。最重要的事情是要把注意力放到要解决的问题上，而不是 C++ 语言的技术方面。如果有一位有经验并且对人没有偏见的朋友或同事帮助你学习一种语言，那总是会更容易些。

CA: C++ 发展的下一步是什么？

BS: 工具/环境和库的设计。我希望看见 C++ 有增加的编译器和链接器，对于一个中等长度的 C++ 程序，在对几个函数局部变化之后，重新编译、重新链接的合适时间应该是 2s；我希望看见这样的浏览器和分析工具，它们不但知道句法，而且还知道每个程序实体的类型；我希望看到优化器，它实际上注意到了 C++ 的结构，并且很好地优化它们，而不是简单地舍弃大部分有用的信息，并把剩余的送到仅能基本上理解 C 的优化器；我希望看见调试器与增加的编译器集成在一起，以使结果近似于一个 C++ 的解释器（我也希望看到一个好的、便捷的 C++ 解释器）。这些都不是科学幻想。事实上，我已经看见了我所建议的大多数内容的实验版，甚至更多，但我们仍然困扰于第一代的 C++ 环境和工具中。

CA: 毫无疑问，C++ 的大部分都是建立在 C 的基础上的（像我们都熟悉的短语“越接近 C 越好，但不能过度”，这句话类似于爱因斯坦的名言“越简单越好，但不能过度”），但是 C 的兼容性确实是一种挑战，而且将要实施的 ISO 标准的进化导致了大量的折衷方案，那么在你的记忆中，主要的折衷方案是什么呢？

BS: 我想大部分的折衷方案都早在标准委员会的会议上就提出来了（看我对下一个问题的回答）。C++ 特有的主要特征，如类、名字空间、模板和异常，被一个愿望束缚着，那就是要能够产生非常紧凑的和高效的代码，并且可以同其它语言编写的代码共存。

事实上，我们可以看到大部分的折衷方案都是被“零开支原则”所左右的，这个原则意味着你不用的任何特色都不占用任何的时间和空间。这使 C++ 成为一种可实行的系统编程语言，从而避免使 C++ 进化为一种更方便的编写自娱例程的工具，而不是平常所用的编程工具。

CA: 暂且假设 C 的兼容性已经不成问题，在你看来，C++应会有什么不同？

BS: 这确实不是一个真正清晰的问题，因为“越接近 C 越好，但不能过度”是基本的设计目的，即向 C++的用户传递有限的技术利益，这不只是一个政策上或商业（广告）上的决策。假如 C 不存在或者存在但不能充分满足我的需要，我就会发现另一种语言和 C++兼容。在设计方面我没有看到优于 Algol 家族语言的另一种语言，而且，发现和维持适度的 C 兼容性的工作没有被低估。C 的兼容性除了是最重要的内容之外，它同时还是最难决定和最难实行的。

虽然，我在许多方面并不喜欢 C，但假如有一种语言没有那些特征，却能像 C 一样高效、灵活、可用，我一定会选择这种语言与 C++相兼容。如，我认为 C 的声明句法就是一个失败的实验，而且一般说来，C 的声明句法过于宽松。特别应注意的是，C++已经去掉了“隐式整型变量规则”：

```
static T;
```

不再是有效的 C++语句，如果你真想声明一个整型变量，必须写成：

```
static int T;
```

许多这样的细节给编译器设计者和现实中的 C 及 C++代码的临时读者制造了不必要的困难（和上面提到的小例子相反）。

我不喜欢预处理器是众所周知的事，在 C 编程中，C++ 是必不可少的，它在传统的 C++ 执行中仍是非常重要的，但它是一个“雇佣工”，因此大部分技术都依赖于它。我的长期目标就是使 C++ 多余，因此，我不梦想废除它直到它真正地成为多余，而这还没有发生。模板、const、inline 和名字空间使大部分宏的用途都成为多余，我们还没有广泛有效的可供选择的方法来代替 #ifdef 的若干普通应用（到目前为止）。预处理器是导致缺乏 C 程序开发环境的主要因素之一，事实上，程序员看到的源文本并不是编译器看到的文本，这是一个临界障碍。我认为该是认真考虑在 C++编程中禁用宏的时候了。

我也发现对于大部分用途来说 C 的数组太低级了，但是，我并不认为 C 的成功是偶然的，它过去是、现在还是在足够多的方面优于任何的替代者，是许多工程的最佳选择，当然，选择 C++时除外。除了 C++编译器缺乏有效性之外，我从来未发现任何理由说对于一个应用 C 是比 C++更好的选择。如果有人发现 C++中有一些非 C 的特性使其不适合某个项目，那么在那里就可以不用它了。

CA: 你承担的 Java 革命的责任是什么？

BS: 什么是 Java 革命？Java 至少有两点与众不同的地方：Java 是相当标准的现代化编程语言，在表面上与 C++有几分相似；而且当提示通过浏览器把代码下载到别人的计算机上的时候，它是一个相当有趣的系统。后者提出了困难的、有趣的而且重要的问题，如果使用 Java 和 Javascript 时这些严重的安全问题能够得到解决，这可能是非常重要的，即使这个安全漏洞还没有被弥补上，它可能也很重要，因为无论怎样，似乎大部分人一般不在乎安全性的问题。

我猜你会问我关于 Java 作为一种编程语言的有关问题以及它与 C++的关系，但是对于这

个问题我不愿说太多，因为不基于有意义的用户体验去比较两种语言是不公平的。根据你前面的问题，我要指出，Java 当然不是我要设计的语言，即使我没有兼容性的约束。

如果人们坚持要比较 C++ 和 Java 的话——就像他们似乎要做的那样——我建议他们去彻底读一下 D&E，看看为什么 C++ 如此这般。根据设计标准考虑一下这两种语言，C++ 和 Java 的不同点不只是表面上的，而且不是每一个优点都会集中到同一种语言中。

CA: 1989 年 12 月你在 X3J16 的就职演说中，提到了如果委员会花了 5 年以上的时间去提出一个标准，那它就失败了，但似乎总共花了几乎 9 年的时间才使得一切就位，对此你有什么评价？

BS: 我只能先想一些“遁词”（意指回避问题的话）了，因为我认为委员会做了一件非常伟大的工作，尽管它花了太长的时间去做。我严重地低估了在令人难忘的各种各样的委员之间达成一致所花费的时间。

然而，回首这几年，就是说到 1995 年 3 月，当我的 5 年任期期满的时候，每个主要的语言特征和每个主要的标准库部分都在适当的位置。如果不是发生了一些用模板编译的麻烦事，如果我们想获得一致和谐的观点，我们可去掉用于润色及 ISO 规则的最后 3 年时间。

我过去曾希望发生而且确实发生了的一件事情是，在标准完成的很久以前，对于标准的执行就已经开始了。许多标准化努力取得的好的成果已经在 C++ 程序员的掌握之中。

CA: 标准的存在将如何改变 C++ 社区和你本人？

BS: 对于社区来说，将会是更好的稳定性、更好的编译器、更好的工具、更好的库、更好的教学材料以及更好的技术。对于我来说，最终我有机会以我想用的方式使用 C++，而没有被对标准工作和语言设计的关注所分散精力，重要的且有趣的主题是编程，而不是编程语言。

我发现有不少人属于两个阵营之一：一些人认为编程语言没有那么重要，只是一种系统的构建程序（许多 C 程序员都这样认为）；而另一些人认为只要在一些非常特殊的语言技巧方面是正确的，编程语言就可以为他们创造奇迹（奇迹从未出现，所以他们把所有的努力都放到设计完美的编程语言上）。我属于第三阵营，我知道一种好的语言对程序员个人以及一个团体都是最有帮助的，但我也知道大多优良的代码是用为人诟病的语言编写的，而不是用号称伟大的语言编写的。

另外最为重要的是程序员对于要解决的问题和解决问题所需技术的理解，一种程序设计语言能够帮助你清晰地表达你的想法，并可以通过提供合适的框架帮助你澄清接近正确的想法。我认为 C++ 比目前任何一种语言应用的范围都广泛，如果人们愿意花时间去学习那些由 ISO C++ 提供的特征产生的技术的话，C++ 会变成一个更好的工具。然而，如果没有正在做什么和如何做的清晰思路的话，无论你选择何种语言来编程，你都会迷惑。